GETTING WHAT YOU WANT FROM THE

TRS-80 MODEL 100®

BASIC PROGRAMMING FOR BUSINESS

by E. PAUL CONE

CONTENTS

Introduction

With this book and your Model 100 you can learn how to write BASIC programs to solve your problems, complete your tasks, and get on with your business. The Model 100 lets you do this anywhere, at any time, with its built-in BASIC programming language. This truly remarkable tool gives you the power to get what you want from your Model 100.

How This Book Is Organized

Getting What You Want from the TRS–80, Model 100: BASIC Programming for Business is a complete course in BASIC programming. The book is divided into seven sections, each designed to give you the knowledge that you'll need to write and understand BASIC programs:

CHAPTER 0 tells you what you need to know before you can learn to write BASIC programs. This chapter goes over the nitty-gritty details, including getting BASIC up and running, and using the keyboard to enter commands. Even if you have been using the Model 100's built-in programs (such as the ADDRSS computerized address book, or the TEXT word processor) this chapter contains lots of new information that applies to BASIC programming.

CHAPTERS 1–5 are the core of the book and teach you how to write programs. Throughout the book each new topic is illus-

trated by sample programs and explained using plain-English analogies and examples from everyday life. Jargon is avoided wherever possible and explained wherever it is necessary.

The sample programs start off simply but build rapidly on what you learn as you go along. For instance, early examples include a simple program to add up travel expenses, while programs later in the book include such things as a Mailing List program that can print mailing labels in either alphabetical or zip code order, and an Order Entry program that prices each line of an order and prints an order confirmation—automatically.

There are sections that show you how to organize a problem so that you can write a BASIC program to solve it. The book even reveals the *Seven Rules of Programming* to help you over the most important hurdles. Of course, each chapter leads you in easy-to-follow steps toward your goal of learning to write BASIC programs and all the sample programs are explained one line at a time.

CHAPTER 6 covers the Model 100's ability to communicate information over the telephone lines (called telecommunications) and explains how to send and receive information for use by your BASIC programs. Chapter 7, on Model 100 graphics and sound, shows you how to create eye-catching displays and printouts to inform and influence others.

CHAPTER 8 is devoted to helping you find errors in your own programs. Chapter 8 also includes a section on how to use the Model 100's TEXT word processor to make program writing easier.

CHAPTER 9 provides the information that you will need if you have the optional Disk/Video Interface. With it, you can apply what you've learned in this book to write BASIC programs that use diskettes as a storage medium.

APPENDIX A contains the ASCII Character Code Table.

APPENDIX B lists the *Seven Rules of Programming*.

All of the sample programs in the book will run on the 8K version of the Model 100. Of course, they'll also run on the 16K, 24K, and 32K Model 100s.

Be sure to read the owner's manual before using your Model 100. This information is rewritten in this book only when it is needed to explain the techniques used to program the Model 100.

The information in *Getting What You Want from the TRS–80, Model 100: BASIC Programming for Business* teaches you how to use the features of the Model 100 programming language. The sample programs are examples of how BASIC language commands can be used on the Model 100. But the material covered in this book and the sample programs should not be used as a guide to managing your business or financial affairs. That is beyond the scope of this book.

How to Use This Book

This is a hands-on course in BASIC programming. In Chapter 1 you will begin to type programs into your Model 100. When the original programs were typeset for this book, some of the individual program lines were too long to fit on one line of the page. For this reason, you may see sample program lines that are reproduced in this book differently than they will appear on the screen of your Model 100. For example, when you get to program line 280, on page 17 of Chapter 1, it will look like this:

```
280 LINE INPUT "ENTER 'R' TO RERUN, 'E' TO
END...";X$
```

Even though it is reproduced on two lines, it's really only one program line. This may not make too much sense to you right now, but it will make sense to you when you get to Chapter 1.

In addition, as you read the book you will notice that when BASIC commands are first introduced and explained they are in boldface, and when new computer jargon is introduced it is underlined. When new commands are introduced, the command tails (the part that the programmer fills in) are printed in italics.

Good luck, and happy programming.

Chapter 0

The BASIC Basics

You're ready to learn BASIC programming on your Model 100. If you're anything like me, you'll want to jump right in and start *programming*. That's the right attitude to have because BASIC programming is simple.

Starting from Zero

But there may be some things you need to know before you can really get started learning BASIC. This book begins with Chapter 0 because Chapter 0 tells you what you need to know before you can learn what you want to know. It's a guided tour of the Model 100 and a reference you can turn to later to refresh your memory.

Before you can start to program, you need to be familiar with the computer that you are programming. Chapter 0 contains some important information about the Model 100: how to turn the computer on, and how to start BASIC, and how to type in BASIC commands at the keyboard. Typing BASIC programs and storing what you've done will be covered in Chapter 1.

If you already have some experience with the Model 100 and BASIC programming, just skim this chapter. If this is your first turn at BASIC on the TRS–80 Model 100, then Chapter 0 is the right place to begin. Chapter 0 starts with the nitty-gritty, but it picks up speed rapidly. One thing that can slow everyone down is computer jargon, and we have a few words to say about that.

Buzzwords, Slang, and Jargon

There are a lot of special terms used in computing. This collection of language—the buzzwords, slang, and jargon of the computer world—is one of the biggest obstacles to get over when you first try to enter the world of computers and computer programming. One goal of this book is to help you get beyond the computer-jargon obstacle course. Before you finish the book you'll know the meaning of words such as byte and RAM. These terms will all be explained when you need them. But right now we'll explain computer jargon itself.

Some computer jargon comes from the field of mathematics, some from the field of electrical engineering, and some seems to have come out of thin air. But one thing is true of all computer jargon: Computer jargon is often just complicated language for simple ideas. This book avoids computer jargon whenever possible, and when a new technical word must be used, we explain it as simply as we can.

Let's get started.

Getting Started

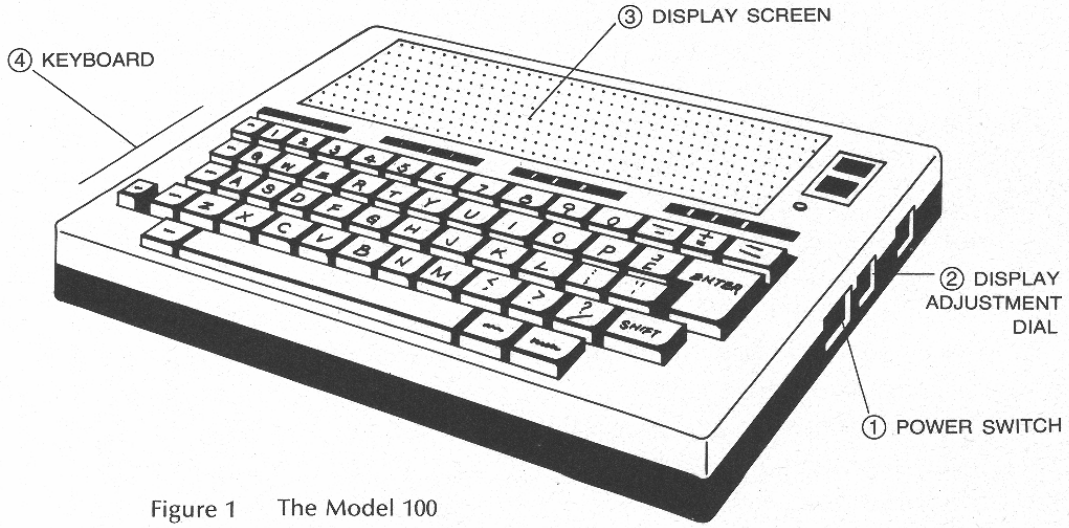


Figure 1 The Model 100

Figure 1 shows the location of: (1) the power switch; (2) the display screen adjustment dial; (3) the display screen; and (4) the keyboard. You need to know how to get started using BASIC. This section shows you how.

Before you can use the Model 100, you have to turn it on. Even this simple act is not as obvious as you might expect. To turn the power on, you must find the power switch and then slide it toward you. With the power on, your Model 100 display screen will look something like this:

```
Jan 01, 2001 Mon 24:00:00 (c) Microsoft
BASIC      TEXT       TELCOM     ADDRSS
SCHEDL     —·—        —·—        —·—
—·—        —·—        —·—        —·—
—·—        —·—        —·—        —·—
—·—        —·—        —·—        —·—
Select:                 12345 Bytes free
```

Figure 2 The Main Menu

If your screen looks blank or the image is not clear, try turning the display screen adjustment dial (located on the right side of the Model 100 behind the on/off switch) forward or backward until you can see the display. If the display screen is not adjusted properly, the screen can appear blank—as if the computer were not on. So turn the screen adjustment dial until you see something that looks like the example above.

Now that you have turned on the Model 100 it will stay on, right? Wrong. In this matter the Model 100 has a mind of its own. The Model 100 will turn itself off automatically if you don't press any keys for ten minutes. This is to conserve the power in the batteries. This makes sense but if you don't know what's going on you may think that your computer has died. If the Model 100 turns itself off, you can turn it on again by sliding the on/off switch to the *off* position and then back again to the *on* position. Let's take a look at the main menu which is shown in Figure 2.

Main Menu

When you turn the Model 100 on, it displays a screen full of information called the main menu. A menu, in computer jargon, is a list of choices, just as a restaurant menu is a list of choices. The Model 100 main menu displays the date, the day, the time, and a copyright notice. Except for the copyright notice, these features are explained in full in this chapter. The last line shows how much memory is left in the Model 100. We will be talking about memory later.

What we are interested in right now are the second and third lines of the main menu. These are the choices that you can make from the main menu. When the Model 100 displays the main menu, it's as if the Model 100 is asking you, "What would you like to do?":

```
BASIC   TEXT    TELCOM  ADDRSS
SCHEDL  —.—     —.—     —.—
—.—     —.—     —.—     —.—
```

These five words, BASIC, TEXT, TELCOM, ADDRSS, and SCHEDL, are the names of five programs that are built into the Model 100. They are ready for you to use as soon as you turn on the Model 100. We've listed them here with brief descriptions:

- BASIC is the BASIC programming language, the subject of this book.
- TEXT is the Model 100 word processing program. We talk about it in Chapter 8.
- TELCOM is the Model 100's telecommunications software. Telecommunications is computer jargon for two computers talking to each other over the telephone lines and is covered in Chapter 6.
- ADDRSS is the Model 100 computerized address book, not addressed by this book.
- SCHEDL is the Model 100 appointment book, but it doesn't fit into the schedule for this book.

Cursor Movement Keys

Here's how to make a choice from the main menu. You can't just answer the main menu's implied question by saying, "I'd like to use BASIC, please." You have to use the cursor to point to the word BASIC. The cursor is the gray rectangle that will appear behind one of the main menu words. You can move it with the cursor movement keys.

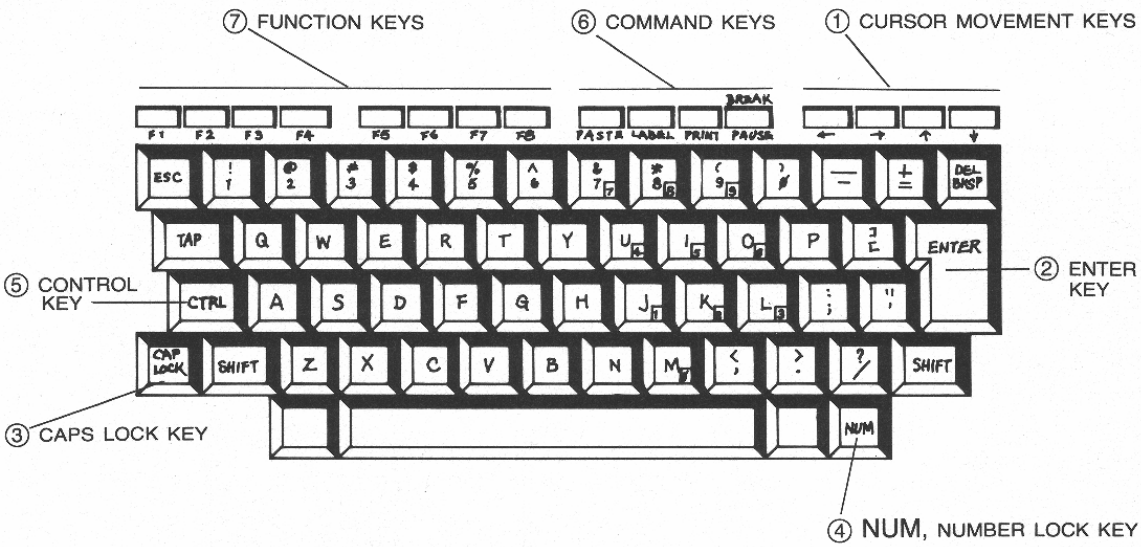


Figure 3 The Keyboard

The cursor movement keys are the four small, black rectangular keys in the upper right-hand corner of the keyboard (#1 in Figure 3). There is an arrow printed under each key. When the key is pressed, the cursor moves in the direction in which the arrow is pointing. With the main menu on the screen, press each of the cursor movement keys and see how they work.

To get ready to use BASIC, move the cursor until it is behind the word BASIC. Then press the **ENTER** key. The **ENTER** key (#2 in Figure 3) is the Model 100's equivalent of a typewriter's RETURN key. But on a computer the **ENTER** key is not exactly the same thing as a typewriter's RETURN key. As we'll see later, the **ENTER** key is used to indicate our wishes to the

computer. We'll explain more about the **ENTER** key later when it is time to type in BASIC commands.

When you press the **ENTER** key, the screen should display this message:

```
TRS-80 Model 100 Software
Copr. 1983 Microsoft
1234 Bytes free
Ok
```

You'd never know it by reading this message, but when you see it the Model 100 is ready for you to use BASIC. The Ok is BASIC's way of indicating that it is ready for you to start writing commands. You can always tell if BASIC is having any problems because the Ok will not be displayed. When BASIC's Ok, we're Ok.

The blinking gray square beneath the Ok is the cursor; it is the size of one character and shows you just where on the screen you are.

With the Ok and the blinking cursor we're now ready to write our first BASIC commands.

BASIC has two modes: command mode and program mode. This needs some explanation. A mode is jargon for the way that BASIC responds to your commands. In command mode BASIC responds to your commands as soon as you enter them. In program mode BASIC only carries out your commands when you run a program. In this chapter we'll see how to enter commands in command mode. Then in Chapter 1, we'll start using program mode.

Entering Commands

What is the nicest thing about computers? Is it that they solve problems, work fast, are super calculators, or do your income tax? No, it's none of these things. The nicest thing about computers is that they *always* do what you tell them to do. Even your dog won't always do that. To get a computer to do what you want you can give it a command by typing the command at the keyboard. BASIC has a large number of commands, and we'll introduce them as we

need them.

The Model 100 responds to one command at a time as soon as you can type it in and press the ENTER key. We're giving the commands. Let's begin by telling the Model 100 to display the sentence:

```
You're the boss.
```

To do this, we have to use our first BASIC command, the **PRINT** command. Type in this line, exactly as you see it here:

PRINT "You're the boss!"

Now press the ENTER key. The **PRINT** command that you entered instructs the Model 100 to display the text that you typed between the quotes:

```
You're the boss!
```

When you pressed ENTER one of two things happened:

- The screen displayed the sentence:

```
You're the boss!
```

```
TRS-80 Model 100 Software
Copr. 1983 Microsoft
6583 Bytes free
Ok
PRINT "You're the boss!"
You're the boss!
Ok
```

Figure 4 The PRINT Command

and the screen looked like Figure 4. After it follows your command, BASIC also displays the Ok and the blinking cursor again to let you know everything is Ok.

- The screen displayed the strange message:

```
?SN Error
```

This cryptic phrase is called a BASIC error message. In this case it was caused by the mistyping of the BASIC word, PRINT (see Figure 5).

```
TRS-80 Model 100 Software
Copr. 1983 Microsoft
6583 Bytes free
Ok
PRNIT "You're the boss!"
?SN Error
Ok
```

Figure 5 The Syntax Error Message

Error Messages

We all make mistakes. BASIC is so sure that we're going to make mistakes that it has built-in error messages to display when we mess up. The message **?SN** is one of these built-in error messages. **?SN Error** is a syntax error message. SN is an abbreviation for SyNtax.

BASIC expects your commands to be in a very specific form, called syntax. If you don't type each command exactly the way BASIC expects it to be, even if a single punctuation mark or a single letter is missing or misplaced, BASIC won't understand you and will display the syntax error message:

```
?SN Error
```

When you get a syntax error message, just retype the command and enter it again. This may be very frustrating at first, but as the BASIC commands become familiar to you you'll make fewer syntax errors. You'll learn more about error messages in Chapter 8, which includes a complete list of Model 100 error messages.

CLS Command

Here's another very handy command. Type:

CLS

Press ENTER. The screen will go completely blank and then display the Ok and the blinking cursor. That's because **CLS** is the **CL**ear **S**creen command. Whenever you enter **CLS** the display screen will be wiped clean.

In the paragraphs above we told you to type in a command and then press ENTER. BASIC knows that it's time to follow your command. From now on in this book, when we write (ENTER) it means the same thing as Press ENTER.

CLS (ENTER)

means "type CLS and then press ENTER."

The CAPS LOCK Key

You probably noticed that the BASIC commands are in all capital letters. To save you the trouble of holding down the shift key while you type commands the Model 100 keyboard has a **CAPS LOCK** key. When this key (#3 in Figure 3) is pressed down, all the letter keys on the keyboard will produce CAPITAL LETTERS, LIKE THIS. AS SOON AS YOU PRESS THE **CAPS LOCK** KEY AGAIN, the keys will produce lower case letters, like this. Try it.

Beware of the NUM Key

When you type on the Model 100 keyboard beware of the **NUM** key. The **NUM** key is sort of like a CAPS LOCK key. Why the one key is called CAPS LOCK, and the other key is called merely **NUM,**

without the LOCK, is anybody's guess. When the **NUM** key (#4 in Figure 3) is pressed down, the U, I, O, J, K, L, and M keys become number keys, and these letter keys become a calculator-type keypad for typing numerical data.

But watch out. With the **NUM** key depressed, typing a command such as CLS will produce the nonsense C3S. If you're not aware of this, you'll get syntax errors until you go crazy, or until you notice that the **NUM** key has been pressed down.

More About the Command Mode: DATE$, DAY$, and TIME$

You can do more in the command mode than just write messages like *You're the boss* and clear the screen. The command mode is also used to give the Model 100 commands that normally would not be written in a program. **DATE$, DAY$,** and **TIME$** are all Model 100 BASIC commands. Remember that the top line of the main menu displays the date, the day, and the time (and the copyright notice):

```
Jan 01, 2001 Mon 24:00:00  (c) Microsoft
```

If you're flying to Europe on the Concorde, you may decide to set your Model 100 for Paris time. You would use the **TIME$** command to reset the Model 100's clock to display Parisian time. Here's how these commands work.

DATE$

You can set the date with the **DATE$** command. The **DATE$** command follows this format:

DATE$ = "month/day/year" (ENTER)

For example, to set the date for June 24, 1985, type:

DATE$ = "06/24/85" (ENTER)

The dollar sign, equal sign, quotation marks, and slash marks

(/) are required for the command to work. And even though the month is displayed on the top line of the main menu as a word:

```
Jan 01, 2001 Mon 24:00:00  (c) Microsoft
```

it is entered in the command as a two-digit number (such as 01 for January, 06 for June, and 12 for December).

DAY$

To set the day of the week, use the **DAY$** command:

DAY$ = "day" (ENTER)

For example, to set the day to Monday, type:

DAY$ = "Mon" (ENTER)

The **DAY$** command recognizes these abbreviations: Mon, Tue, Wed, Thu, Fri, Sat, and Sun.

TIME$

The time in the Model 100 is displayed in *24-hour notation*.

In 24-hour notation the time is displayed as:

hours:minutes:seconds

Before noon, 24-hour notation is easy. For example, 9:15:00 is 9:15 A.M. But to display P.M. time in 24-hour notation, 12 must be added to the *hour*. For example, 21:15:00 is 9:15 P.M., and 13:15:00 is 1:15 P.M.

Thus, if you want to set the Model 100 clock to 9:15 A.M., you type the **TIME$** command exactly as shown here:

TIME$ = "09:15:00" (ENTER)

The dollar sign, equal sign, quotation marks, and colons (:) are required. To set the time to 9:15 P.M., type:

TIME$ = "21:15:00" (ENTER)

Remember, 9:15 P.M. is 21:15:00 in 24-hour notation.

The MENU Command

Well, you're really letting the Model 100 know who's boss. But it's time to leave BASIC and return to the main menu to see if our **DATE$, DAY$,** and **TIME$** commands worked. To get back to the main menu, type:

MENU (ENTER)

When the main menu is displayed again, see if you have set the date, day, and time the way you wanted to. If not, return to BASIC and try again. Once you have set them correctly, the Model 100 main menu will display the date, day, and time perpetually.

The NEW Command

In the next chapter you will begin to type programs into your Model 100. Before you start to type a program you should enter the **NEW** command:

NEW (ENTER)

This command clears the Model 100's memory and prepares it for the *new* program that you are going to type in. Before you enter the **NEW** command you should SAVE any program that is in memory, since the **NEW** command will wipe out the program that is currently in memory.

That's it for commands, for now. If you think there's more to BASIC programming than this, you're right. There's a lot more, and we'll start right away. In the next chapter you'll type in your first BASIC program and learn how it works. From here on, the pace quickens. Let's do it.

Chapter 1

Your First BASIC Program

In Chapter 0 you entered commands. In this chapter you will actually enter your first program and take it for a test run. There's some computer jargon. To run a program is to use the program on the computer, just as you run a car or a machine. You'll also learn how to correct any errors in the program. (there are *always* errors). There's a great deal of information in this chapter, so take it one step at a time. There are resting spots along the way where you can bail out for a break.

If you know a little about BASIC already, you might think we're touching lightly on some points. You might be right. This chapter introduces programming concepts, and everything that is covered here is discussed in more detail in later chapters.

Our first stop along the way is a look at the BASIC programming language itself.

The BASIC Programming Language

The Model 100 can only do one thing at a time and it doesn't even pretend to do two things at once. It will plod along from one task to the next, doing exactly what it's told to do when it's told to do it. Just as it followed your commands in Chapter 0, it will follow your programs in Chapter 1.

```
10  ' TRAVEL EXPENSE REPORT
20  '12 AUG 83
30  CLS
40  PRINT " TRAVEL EXPENSE REPORT--"
50  PRINT " RESPOND TO EACH REQUEST WITH
YOUR"
60  PRINT " TOTAL TRIP EXPENSE FOR THE
CATEGORY"
70  PRINT            'INPUT SECTION
80  INPUT "    LODGING EXPENSE";LX
90  INPUT "        FOOD EXPENSE";FX
100 INPUT "AIR TRAVEL EXPENSE";AX
110 INPUT "       OTHER EXPENSE";OX
120               'CALCULATION SECTION
130 TX=LX+FX+AX+OX
140               'OUTPUT SECTION
150 CLS
160 PRINT "    LODGING EXPENSE=";
170 PRINT USING   "$$####.##";LX
180 PRINT "        FOOD EXPENSE=";
190 PRINT USING"   $$####.##";FX
200 PRINT "AIR TRAVEL EXPENSE=";
210 PRINT USING"   $$####.##";AX
220 PRINT         OTHER EXPENSE=";
230 PRINT USING   "$$####.##";OX
240 PRINT "----------------------------"
250 PRINT "       TOTAL EXPENSE=";
260 PRINT USING "$$####.##";TX
270 PRINT "PRESS <PRINT> KEY FOR HARD
COPY, OR"
280 LINE INPUT "ENTER 'R' TO RERUN, 'E' TO
END . . . ";X$
290 IF X$="R" OR X$="r" THEN GOTO 30
300 IF X$="E" OR X$="e" THEN CLS: END
310 CLS: END
```

Figure 6 Travel Expense Report Program

For this reason, writing computer programs for the Model 100 can be very simple. But the Model 100 understands a different language. If you want to communicate with it, you've got to speak its language—and the language is BASIC. Millions of people have learned to speak the BASIC language to millions of personal computers all over the world. And you will too.

The best way to learn BASIC is to sit down with the Model 100 and type in each program in this book. Run them on the Model 100 and see how they work. Read the description given of the BASIC commands used and understand what each part of each program is making the computer do.

This method really works. If you make a mistake while you're trying to get a program to work, consider it a blessing. This isn't just a soothing phrase. To get the programs typed in correctly you really have to try to understand what each line does. It's not just a typing assignment. Making a mistake helps force you to understand what you're doing wrong and how to correct it. And getting a program to work correctly is a very satisfying experience.

The first BASIC program in this book, Figure 6, is a Travel Expense Report program. It introduces the fundamental concepts of a well-written BASIC program. Even if you want to get going, don't try to type in the entire program right away. Follow along with the chapter. There's so much to learn at first that we've combined the typing in of each line of the program with learning about what is happening. But we have a lot of fundamentals to cover before even one line is typed in. Time laying the foundation of programming now is time well spent.

Look at the program, the first six lines of which are reproduced below:

```
10 'TRAVEL EXPENSE REPORT
20 '12 AUG 83
30 CLS
40 PRINT " TRAVEL EXPENSE REPORT --"
50 PRINT " RESPOND TO EACH REQUEST
WITH YOUR"
60 PRINT " TOTAL TRIP EXPENSE FOR THE
CATEGORY"
```

Each instruction is written on a numbered line. Each line of a BASIC program is called a line of code. Each line must begin with a number, called a line number.

The BASIC language is like English spoken by a New Englander who, when asked a long, philosophical question, responds by saying only, "Yup." That means that there are no unnecessary words in the BASIC language.

Each word in each line has a function. As with commands, words such as PRINT or CLS have special meanings. The line numbers don't have to be consecutive and they don't have to start at 10, but they do have to proceed in increasing order.

Most programmers find it a good idea to number program lines in increments of ten to allow insertion of new lines when mistakes are discovered or additions need to be made. By now you should have guessed, correctly, that computer programming has a lot to do with making and finding mistakes. The program examples in this book are all shown with line numbers in nice increments of ten, but I made plenty of mistakes when writing them.

Program Lines vs. Commands

The line numbers distinguish a BASIC program from simple one-line commands. What does that mean? In Chapter 0 when you typed in the CLS command you didn't use a line number. You just typed CLS, and pressed the ENTER key:

CLS (ENTER)

The screen cleared immediately. That's because we were in the command mode. But when you type a program line, you first type the line number and then the command, followed by (ENTER). For example, to type in a program line that clears the screen, type:

10 CLS (ENTER)

When you press ENTER, notice that nothing happens. The screen does not clear. That's because when BASIC sees the line number in front of the command it knows you are in program mode, and that line **10**, as short as it may be, is a BASIC program, not

just a command. If you want the computer to run a program, enter the command **RUN**, on the next line. If you type the program line and then type **RUN**, the screen should clear. The entire process would look like this:

10 CLS (ENTER)
RUN (ENTER)

As with commands, BASIC expects program lines to be entered with the correct syntax. If you misspell CLS, the computer will display an error message when you try to run the program. When BASIC displays an error message in a program it will also display the line number where it found the error. This is a nice feature because it can help you find mistakes. This is what the screen will look like if you name a syntax error in our one-line program:

```
10 CSX (ENTER)
RUN
?SN Error in 10
Ok
```

Try entering this misspelled program line and see what happens (notice that CLS is misspelled CSX). You may as well try it now when you know the cause, because it is bound to happen later when you don't. When BASIC finds an error in a program line it stops immediately and displays the error message without going any farther. That's why the screen didn't clear in our example above. When you enter the line correctly and then enter **RUN** the screen clears.

In computer jargon, the word for an error in a program is bug. That's sort of poetic. At least it's more poetic than *?SN Error in 10*.

When a bug causes the program to stop unexpectedly it is called bombing out, or crashing. This jargon conveys the feeling you get when your program flops. You may have some words of your own for it. Fixing bugs is called debugging, and we've devoted an entire chapter to it, Chapter 8.

More About Program Lines

We said that each program line always contains one BASIC command, but that was a simplification. More than one command can be written on a line. For example, the two PRINT statements below are written on two separate program lines.

```
10 PRINT "This"
20 PRINT "That"
This
That
```

These two print statements can be combined on one program line, and still produce the same results. To do so, separate them with a colon (:) and omit the second line number:

```
10 PRINT "This": PRINT "That"
This
That
```

There's another method of writing program lines that you may see in magazines. Programmers sometimes *omit the spaces* in program lines to save space in memory. The line used in the previous example could have been written like this:

```
10 PRINT"This":PRINT"That"
This
That
```

The results of this compact line are the same, but writing program lines in this way is not recommended for beginning programmers. Now if you see programs written this way in magazines, you will know what's going on. But don't do it yourself. It makes the program harder to read and it makes mistakes harder to find.

The Travel Expense Report Program

We took a big detour from our trip through the Travel Expense Report program. Now we'll return to it, going over each numbered line as if we were the computer and examining each line as you type it in to see what the instructions mean.

The Remark Lines

Lines 10–20 are remark lines. A remark is just what the name implies; remark lines allow you to write lines that have no effect on the BASIC program. They are for your benefit. Anything to the right of the word REM or the apostrophe character (') BASIC ignores.

A well-written program contains enough remarks to identify the program's major sections and peculiarities, even if you do not expect anyone else to see it. When you try to modify programs months after they were first written, a few well-placed remarks can save many hours of hair-pulling. Even line 10:

```
10 'TRAVEL EXPENSE REPORT
```

can save you the trouble of reading through the program to remember what it was for. You won't really appreciate the remark in line 10 until you have twenty programs on your desk and you have to find one of them in a hurry . . . for your boss . . . on Friday at 5:02 P.M.

CLS

Line 30 uses the command CLS to clear the screen. This erases any information that was displayed before the program began execution and cleans up the screen for the current display, described next.

Figure 7 is a copy of the display created by lines 30–110.

```
TRAVEL EXPENSE REPORT --
RESPOND TO EACH REQUEST WITH YOUR
TOTAL TRIP EXPENSE FOR THE CATEGORY

   LODGING EXPENSE? 200
      FOOD EXPENSE? 100
AIR TRAVEL EXPENSE? 300
     OTHER EXPENSE? 75
```

Figure 7 Input Screen

PRINT

Lines 40–60 use the PRINT command that we saw briefly in Chapter 0. The PRINT command follows the format:

PRINT *"text you would like to display"*

The text between the quotation marks is displayed on the screen. Any text that is to be displayed must always be written between the quotation marks. If there is no information in quotes following the PRINT command, as in line 70, only a blank line will be displayed. Line 70 uses this to separate the top and bottom of the display, as shown in Figure 7.

LIST

How're you doing? You can see how well you're doing at entering the program lines by using the **LIST** command. When you type:

LIST (ENTER)

the Model 100 will display on the screen all the program lines you've entered so far. This is called listing the program. Try to list whatever you have typed so far. You should see the lines below appear on your screen.

```
10 'TRAVEL EXPENSE REPORT
20 '12 AUG 83
30 CLS
40 PRINT " TRAVEL EXPENSE REPORT --"
50 PRINT " RESPOND TO EACH REQUEST WITH
YOUR"
60 PRINT " TOTAL TRIP EXPENSE FOR THE
CATEGORY"
70 PRINT
```

In this example, all the program lines we've typed in are displayed on the screen. This is because we only have six lines and the Model 100 screen can display eight lines. But what happens later when you've typed in the entire program and you list it?

You'd better be able to read fast because the listing moves up the screen line by line and faster than the human eye can follow. Unless you are a super speed-reader you'll need a way to freeze long listings if you need to check the program. To do this use the **PAUSE** key.

Pressing the **PAUSE** key freezes the listing. The **PAUSE** key is one of the Model 100 command keys (labeled #6 in Figure 3, Chapter 0). Pressing **PAUSE** once freezes the listing so you can read it. Pressing **PAUSE** again unfreezes the listing to continue the display.

You can look at any one line in a long program by using the **LIST** and the line number. If you type the following command, for example:

LIST 40 (ENTER)

Line 40 will appear on your screen as shown below:

```
40 PRINT " TRAVEL EXPENSE REPORT --"
```

LLIST

It's much easier to look at program listings when they're printed on paper. When you have a printed listing in front of you, you can use a pencil to correct errors, make notes, and doodle while

you think about what to do next. If you have a printer connected to your Model 100, typing the LLIST command (note the two Ls):

LLIST (ENTER)

will print the listing instead of displaying it on the screen.

Correcting Errors in Listings

By the way, if you list the program and find that you made a mistake, such as:

```
40 PRRRINTT " TRAVEL EXPENSE REPORT --"
```

you can correct it by retyping the entire line, including the line number.

```
40 PRINT " TRAVEL EXPENSE REPORT --"
```

BASIC remembers the latest version of any numbered lines you type. This can be a mixed blessing because if you just type:

40 (ENTER)

BASIC will remember line 40 as . . . nothing.

Saving and Retrieving Programs

If you've come this far and you'd like to stop, you can save the program you are typing until you want to use it again. There are two ways to bail out of BASIC:

- The **MENU** command.
- The **SAVE** command followed by the MENU command.

You recall the MENU command from Chapter 0. It can be used alone like so:

```
MENU (ENTER)
```

If you leave BASIC this way, it brings you back to the main menu while BASIC holds your program. When you return to BASIC the program will be there waiting for you. The Model 100 is a pretty helpful computer in this way.

You can also use the **SAVE** command followed by the MENU command:

```
SAVE "TRAVEL" (ENTER)
MENU (ENTER)
```

This **SAVE** command allows you to store the program exactly as it is, under the name TRAVEL.

TRAVEL is called the filename. The filename identifies what you saved, just like the name you write on a file folder, so you can find it later when you want to use it again. When you use the SAVE command, the name you type between the quotation marks becomes the filename. Filenames may have up to, but no more than, six characters, and must always start with a letter. Chapter 5 has more to say about SAVE commands and filenames.

When you use the SAVE command the Model 100 might display one of several error messages. If you get an error message, check the following list for one of these mistakes:

- Did you use the quotation marks?
- Did you type in a filename? BASIC won't SAVE a program without a filename.
- Did you use a filename that was longer than six characters?
- Did you choose a filename that began with a character other than a letter? Filenames must begin with a letter, never a number or a symbol like @ or &.

When you SAVE a program you don't automatically return to the main menu. But when you then use the MENU command to return to the main menu the name of your program, TRAVEL, will be displayed on the menu along with BASIC, TEXT, TELCOM,

ADDRSS, and SCHEDL. Your Model 100 display screen should now look something like this:

```
Jan 01, 2001  Mon  24:00:00 (c) Microsoft
BASIC  TEXT        TELCOM   ADDRSS
SCHEDL TRAVEL.BA  _._       _._
_._    _._        _._       _._
Select:             12345 Bytes free
```

The filename TRAVEL plus the extension .BA displayed on the main menu tells you that you have a BASIC program stored away.

Loading the Program

You've saved TRAVEL and returned to the main menu, and now you're ready to add to it. Return to BASIC and when you see the Ok, type:

LOAD "TRAVEL" (ENTER)

This is called the **LOAD** command. With SAVE and LOAD computer jargon makes sense for a change.

After you load a program always LIST it to make sure all of the program has been loaded. You don't want to start adding lines to a wrong or incomplete program, do you?

Program Structure

Organizing a program into functional sections is called structured programming. BASIC doesn't require this approach, but you will find using it results in much *better* programs.

When you run a program in BASIC the computer starts with the first line, in this case, line 10. The Model 100 performs any instructions on that line and goes to the next line. It continues from one numbered line to the next, following any instructions until it

reaches the end of the program.

BASIC lets you write any instructions in any order. As long as BASIC can follow your instructions and there are no bugs in the program the Model 100 will do what the program instructs it to do. This is a mixed blessing, because programs written in BASIC can become so convoluted that they are impossible to understand and debug. This is why it is important that you organize your BASIC programs into discrete sections. Remember, when bugs occur it will be easier to check out what is wrong if you keep your programs structured. Chapter 8 has more to say about this, but don't go there yet. We've got plenty to do right here. Well-written programs are divided into three major sections:

- The *input* section accepts data to be used by the program.
- The *calculation* section produces the values to be displayed as the results of the program.
- The *output* section displays the results in a meaningful form.

The three sections of the Travel Expense Report program are clearly labeled by remarks in lines 70, 120, and 140. Let's go through each section of the program and see what they do.

INPUT SECTION

The **INPUT** statement, used in lines 80–110, is BASIC's way of asking for information. The **INPUT** statement asks for a number—an amount, quantity, or value—that will then be used in the program. The **INPUT** statement follows the format:

INPUT *"optional message" ;variable*

The optional message tells you what you are expected to supply. The message is followed by a semicolon, and a future place for the number, a variable name. The reason for the variable name will be explained shortly. First let's look at the **INPUT** statement.

Consider line 80:

```
80 INPUT " LODGING EXPENSE";LX
```

When this line of the program is executed, the screen will display:

```
LODGING EXPENSE?
```

(The question mark is supplied by BASIC.)

When the BASIC program comes to the end of line 80 it stops everything and waits for the lodging expenses amount to be entered. Once an amount has been typed in and ENTER is pressed, BASIC associates the number you entered with the variable LX. Lines 90–110 follow the same format, assigning values to the variables FX, AX, and OX. These symbols, LX (Lodging expense), FX (Food expense), AX (Air travel expense), and OX (Other expense) are all variable names. The variable name gives BASIC a way to keep track of the numbers you supply. The variable name is like a label that BASIC attaches to each number. We'll have a lot more to say about variables in Chapter 2.

Once the BASIC program completes lines 80–110 it has all of the values it will need to add up your trip expense. The input section is finished. It's time to go on to the calculation section.

CALCULATION SECTION

The purpose of a computer program is very often to perform calculations that are repetitive, complex, or prone to error when done by people. Therefore, the program must have lines of instructions that tell the computer how to do the calculation—the calculation section.

When you've supplied the values in lines 80–110, the Model 100 performs the calculation in line 130:

```
130 TX = LX + FX + AX + OX
```

The TOTAL EXPENSE, represented by the variable TX, is equal to the sum of the four values entered in lines 80–110. The method followed by BASIC calculations is very straightforward and similar to the method you would use with pencil and paper. It adds up the values of the four variables to the right of the equal sign and calls them TX.

Assignment Statements and Expressions

Line 130 is a type of assignment statement. An assignment statement takes the form of:

variable = value

It is called an assignment statement because it assigns a value to the variable. In line 130, to the right of the equal sign, there is an expression. An expression is computer jargon, borrowed from math, for a sequence of arithmetical operations. In the well-known 1 + 1 = 2, *1 + 1* is the expression. In line 130 the expression is:

```
LX + FX + AX + OX
```

BASIC uses the symbols shown below to indicate arithmetic operations and evaluates expressions from left to right, according to these specific rules.

Order of Evaluation:

^	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction
( )	Expressions within parentheses are evaluated independently

The BASIC expression:

```
D = W/7*(365/Y)
```

is the same as:

$$D = \frac{W}{7} \times \left(\frac{365}{Y}\right)$$

OUTPUT SECTION

After line 130 adds the TOTAL EXPENSE and assigns the results to the variable TX, BASIC stores the value of TX in the memory of the Model 100. But when BASIC completes line 130 you still don't know the value of TX, the total expense.

Sending results of calculations or other information from a computer program to the outside world is called outputting data. Just as you input your expenses, the program outputs the results. This is accomplished in the output section of the program.

In the output section of the Travel Expense Report, lines 160–260 cause the results of the program to be displayed on the screen. Refer to Figure 8, which shows how this output looks on the screen. The total expense is $675. We could have simply written:

```
160 PRINT TX
```

The computer would then have displayed:

```
675
```

This tells you the "answer." But it doesn't explain what's going on, or give you the opportunity to check the responses you entered in the input section. And even in this rudimentary expense report, you're going to want to know more than just "675."

Just as we put remarks in the program itself to remind us what it's for, we put titles and headings in the displays created by the program to remind us what information is being displayed. When you write programs for yourself it's tempting to believe that you will remember how to use the program in a week, a month, or a year. But if you're like many people who own the Model 100, you're likely to be involved in many demanding activities. Experienced programmers find that they can forget how to use a program that they've written themselves, and for this reason they make their programs user-friendly. When programs explain what's going on they're called user-friendly because, the theory is, like a friend, they help you, the user, along. The Travel Expense Report program does this. The programs throughout this book have many features that make them user-friendly. We'll point them out as they are

used. Now let's get back to the output section and see how we made it user-friendly.

In the output we show the values that you entered for each of the expense categories, along with the total expense, in a neatly formatted screen display.

To do this we send the information to the screen using pairs of PRINT statements, as in lines 160–170. First in line 160 the PRINT statement displays the legend, "LODGING EXPENSE=". Notice the semicolon (;) at the end of line 160. When BASIC finds this semicolon at the end of the PRINT statement, it doesn't skip down to the next line of the display as it normally does. Instead it displays the information from the PRINT statement in line 170 on the same line as the PRINT statement in line 160.

Here's what we mean. From these two lines, without the semicolon:

```
10 PRINT "THE SKY ABOVE"
20 PRINT "THE EARTH BELOW"
```

the output will be:

```
THE SKY ABOVE
THE EARTH BELOW
```

But if we place a semicolon at the end of the first PRINT statement:

```
10 PRINT "THE SKY ABOVE";
20 PRINT "AND THE EARTH BELOW"
```

the output will be:

```
THE SKY ABOVE AND THE EARTH BELOW
```

The Formatted PRINT Statement

Line 170 also tells BASIC how to format the output via a **PRINT USING** command. In this case the instruction,

```
170 PRINT USING "$$#####.##";LX
```

will print the value of lodging expense, LX, using the dollars and cents format with a dollar sign and two digits to the right of the decimal point. Line 170 is your way of saying to BASIC: "Print the value of LX using the dollars and cents format, please."

The four pound signs (#) reserve four places to the left of the decimal point and the two dollar signs ($) cause a dollar sign to be displayed. You can use as many pound signs as you think will be needed (for that expensive round-the-world trip) but for now type the lines just as they are in Figure 6. Lines 160 and 170 together send one line to the display screen of the Model 100,

```
160 PRINT " LODGING EXPENSE=";
170 PRINT  USING "$$#####.##";LX
LODGING EXPENSE= $200.00
```

This process is repeated, two lines at a time, through line 260 until all the information is displayed, as shown in Figure 8.

```
          LODGING EXPENSE=    $200.00
             FOOD EXPENSE=    $100.00
       AIR TRAVEL EXPENSE=    $300.00
            OTHER EXPENSE=     $75.00
                           --------------
            TOTAL EXPENSE=    $675.00
PRESS <PRINT> KEY FOR HARD COPY, OR
ENTER 'R' TO RERUN, 'E' TO END ...
```

Figure 8 Output Screen

Well, you're really speeding along. If you need a break, this is a good spot to rest.

Now that you've come this far you can be sure that you'll make it to the end of the book. You've completed almost an entire program and now it's time to wind it up.

Command Keys

In line 270, after the results of the calculation are sent to the screen, a message is displayed:

PRESS (PRINT) KEY FOR HARD COPY

This message refers the user to another command key, the PRINT key, located above the Model 100 keyboard. By simply pressing the PRINT key, the eight lines displayed on the screen will be sent to a printer, if one is attached. Beware: If no printer is attached and you press the PRINT key, the Model 100 will "freeze" since it is trying to print on a nonexistent printer. If this should happen, press the SHIFT and BREAK keys simultaneously to unfreeze the Model 100.

Branching and Conditional Statements

We see in lines 280–310 that the instructions in a program are not always executed in consecutive numerical order.

```
280 LINE INPUT "ENTER 'R' TO RERUN, 'E' TO END
. . . ";X$
290 IF X$="R" OR X$="r" THEN GOTO 30
300 IF X$="E" OR X$="e" THEN CLS:END
310 CLS:END
```

BASIC contains special statements, called **branching** and **conditional** instructions, that can send a program to any line. This ability to jump from one line in a program to another is an important characteristic of a BASIC program. It is sometimes called passing control from one line to another. The use of branching and conditional instructions will be covered thoroughly in Chapter 3. Let's see how they work here.

After line 270 is executed the program has really been completed. Line 280 could be a simple END statement:

```
280 END
```

This would END the program in an orderly fashion, from the point of view of the computer. But it would leave information from the previous run of the program on the screen and would not give the user the option to rerun the program "automatically." Lines 280–310 make these options possible.

LINE INPUT

The **LINE INPUT** statement follows the format:

LINE INPUT *"optional message"; string variable*

The INPUT statement, used in lines 80–110, asked you to enter *numerical* information at the keyboard. The **LINE INPUT** statement, in line 280, is distinguished from an INPUT statement because it asks the user to enter *alphabetical* information, in this case an R or an E. Look closely at the variable X$, used at the end of line 280.

```
280 LINE INPUT "ENTER 'R' TO RERUN, 'E' TO
END . . .'';X$
```

Do you notice the dollar sign ($)? This dollar sign makes X$ a special type of variable. We should mention here that variables are a very important part of programming. And since variables require so much attention they have acquired entire dictionaries of their own jargon. And right now, when things are getting tricky, who needs more jargon? Unfortunately the next paragraph is filled with both new ideas and new jargon.

When a dollar sign is tacked onto the end of a variable it is called a string variable. String variables store letters or symbols like A, B, C, a, b, c, @, #, $, %, and ¢. These nonnumbers are called strings. There's a lot more to say about variables and strings in Chapter 2. For now, just remember that the LINE INPUT statement in line 280 is asking for something other than a number.

Line 280 asks the user to:

```
ENTER 'R' TO RERUN, 'E' TO END . . .
```

The string that you enter, either 'R' or 'E', is saved by the string variable, X$. If you want to rerun the program, enter the letter 'R' by pressing the R key and then the ENTER key. What would happen then? In the next line, 290, we encounter another new type of program statement, the IF . . . THEN statement, and more jargon.

IF . . . THEN STATEMENT

An **IF . . . THEN** statement is your way of asking the BASIC program to make a decision. When programs make decisions they compare things. The **IF . . . THEN** statement in line 290 compares the letter that you entered in line 280 with the letter 'R'. If you enter the letter 'R' (or small 'r') the program goes back to line 30 and starts again. Here's how it works. The **IF . . . THEN** statement in line 290 takes the form:

IF *(you entered 'R') OR (you entered 'r')*
THEN *(go back to line 30)*

When the BASIC program comes across the **IF . . . THEN** statement in line 290 it pauses to see if X$ equals 'R'. The two expressions in line 290 are *X$ = "R"* and *X$ = "r"*. Since you entered 'R' in line 280, X$ *does* equal 'R'. The expression *X$ = "R"* matches. In computer jargon, when expressions match they are said to be true. What does that have to do with **IF . . . THEN** statements?

If the expression is true, the program looks over to the right of the THEN and follows whatever command you put there. In line 290 the command to the right of the THEN is:

```
290 IF X$="R" OR X$= "r" THEN GOTO 30
```

This GOTO command (pronounced Go To) sends the program back to line 30, where the program starts over again.

If the user enters anything other than 'R' or 'r', the expressions are not true. The BASIC program then ignores the entire line and proceeds to the next line without following the instruction to the right of the "THEN".

When the expressions in an **IF . . . THEN** statement are not true it's a little bit like the Monopoly game instruction: "Go straight to the next program line. Do not pass THEN. Do not rerun the program."

When BASIC ignores a THEN statement like this it is called falling through to the next line. More jargon for you.

In line 300 the variable X$ is again the subject of an **IF . . . THEN** statement. This time if the value of X$ is equal to 'E' or 'e', then BASIC clears the screen (CLS), and ends the program (END).

If neither 'E' nor 'R' is entered, the program falls through to line 310. Line 310 ENDs the program in an orderly fashion. In longer, more complex programs maintaining control like this at the end is important.

Some time-consuming programs even display the message, "PROGRAM ENDED" so that the user won't sit there staring at the screen waiting for something to happen.

After you have typed in the entire Travel Expense Report program run it by typing:

RUN (ENTER)

If you get any error messages, fix the errors. Then save the program with the SAVE command, and return to the main menu with the MENU command.

Function Keys

Now that you are familiar with the basics of your Model 100 and with typing in program lines, here are some shortcuts.

There's a quicker way to get from BASIC to the main menu than by typing in a command. The Model 100 has eight function keys (#7 in Figure 3) which have been programmed to perform certain tasks. For instance, when you are using BASIC you can return to the main menu by pressing function key **F8,** MENU.

The Model 100 really does have many good features. You don't even have to memorize the functions of the function keys. When you press the **LABEL** key (the **LABEL** key is two keys to the left of the PAUSE key) the functions of keys F1–F8 are displayed

along the bottom of the screen. When you receive your Model 100, they may look like this:

```
File  Load  Save  Run  List        Menu
```

This is because function keys F6 and F7 are not preprogrammed (more about this in Chapter 2).

You already know how to use five of these functions. When you press the RUN, LIST, and MENU keys the Model 100 carries out the appropriate command instantly. We'll save the FILE key for Chapter 5.

Using the **LOAD** or **SAVE** keys is a two-step process. To save a program press the **SAVE** key, F3. The Model 100 then displays the command and the opening quotation mark:

```
SAVE "
```

You complete the command by typing the filename and the closing quotes and pressing ENTER, like so:

```
SAVE "FLNAME" (ENTER)
```

The function keys are simply time savers; using them is optional. You press one function key instead of typing the command, but if you feel like typing, go ahead.

So far you've entered the entire Travel Expense Report program in Figure 6, corrected any errors, and run it. The following chapters of this book will slowly increase your ability to use BASIC. Sample BASIC business programs in each chapter dynamically illustrate the topics covered, building on previous information.

The Basic Interpreter

If you'd like to learn a little bit about the technology of BASIC, read this section. If it doesn't interest you, skip directly to Chapter 2.

BASIC is like an onion: Just when you think you've gotten to the center of it you find another layer to peel away. The BASIC onion has three main layers.

BASIC is a foreign language to both humans *and* computers. This may sound strange; but a BASIC program is not actually understood by the computer; it is merely the top layer that you see. On the second layer there is a special computer program within the Model 100, called the BASIC interpreter, that translates our BASIC instructions one step further. And it is the third layer, the translation, that controls the computer itself. We speak BASIC to be understood by the BASIC interpreter. Then the interpreter translates our instructions into very rudimentary instructions, called machine language, to be executed by the computer. Why machine language?

The reason for this is that the computer itself carries the one-step-at-a-time routine to extremes. Computers are *machines*. If we had to write our program in all of the minute steps that the machine executes, we would have a list of numerical instructions like a long length of ticker tape. After all, programming languages are symbolic. The words that a programmer writes in the lines of a program have to be translated into the minute steps that the computer can execute. At one time, programmers had to write their instructions using only 1s and 0s. This was called machine language and looked like this:

```
01011010
01101011
10101010          (a program would contain a
11111111          long list of these 1s and 0s)
10001000
10001111
```

After a quick glance at this, it's easy to see why a language like BASIC was developed with its English language–like commands. A command such as PRINT is a lot easier to remember than a sequence of 0s and 1s like that shown above.

Now, instead of your struggling with machine language, the BASIC interpreter does all that work for you. It translates the BASIC

instructions into machine instructions of 1s and 0s, sends them to the computer, and monitors the entire process.

From your point of view as a programmer you need only remember that the computer manipulates symbols. You write programs using the symbols of the BASIC language, and the interpreter translates your symbols to the machine language code. Of course, at its heart the computer is an electronic device and the entire process is one of a series of flickering voltages.

Chapter 2

Getting Ready to Write a Program: Program Structure and Variables

In Chapter 1 we analyzed our first BASIC program, seeing how it was put together and how it worked. In this chapter we'll begin to see how to organize a problem *before* a program is written. And we'll take a closer look at one of the most important parts of a program, the *variables*.

Getting Organized

As you proceed through this book you'll see that in programming, as in business, the first and most important step is to *get organized* and know what you want to do. As you probably remember, in Chapter 1 the Travel Expense program was divided into three sections:

- An INPUT section that accepts data to be used by the program.
- A CALCULATION section that produces the values to be displayed as the results of the program.

- An OUTPUT section that displays the results in a meaningful form.

In this chapter you'll see one way to go about organizing a program into these sections *before* it's written. When you write your own programs, try to avoid the temptation to just sit down and start typing program lines. Plan, or outline, the program first—or you'll find yourself in big trouble later. This chapter shows you one way to take the giant step from having an idea for a program to writing the numbered program lines that make the idea work.

Although all the programs in this book have been written for you, so that in a sense you don't have to write any programs, following the explanations of how they were written and what each line means will teach you how to program.

This chapter also takes an in-depth look at variables and it introduces some new buzzwords for your burgeoning vocabulary of jargon.

The first program we're going to write is a Printer Sales Report program. This program adds up the weekly sales total for each printer, computes the commission for each one, and displays this information.

One well-known technique to use when designing a program is to draw a diagram of the output as you would like to see it

```
PRINTER    QUANT    COST    SALES    COMM
1          ØØØØ     ØØØØ    ØØØØ     ØØØ
2          ØØØØ     ØØØØ    ØØØØ     ØØØ
3          ØØØØ     ØØØØ    ØØØØ     ØØØ
4          ØØØØ     ØØØØ    ØØØØ     ØØØ

TOTALS     ØØØØ             ØØØØ     ØØØ
```

```
      WEEKLY PRINTER SALES SUMMARY

TOTAL QUANTITY SOLD                ØØ
TOTAL DOLLAR SALES            $    ØØØØ.ØØ
TOTAL COMMISSION              $    ØØØØ.ØØ
```

Figure 9 Display Diagrams: Sales Report Output

displayed on the Model 100 screen. Once you have this diagram you can *work backwards* to create the program that then produces it. This is sort of a "what you see is what you get" approach to programming, and it works.

The Model 100 display screen is 40 columns wide and only 8 lines deep. We have to keep this in mind because only 8 lines of output can be displayed at one time. Figure 9 shows diagrams of how we would like the Model 100 to display the results of the Printer Sales Report program. It was made using the video display worksheet in Figure 90, on page 199.

Writing the First Lines

Armed with the diagrams of the output, we attack the problems of writing the program using a five-step process. These five steps include such words as variables and input section. We warned you that many computer words may impede your progress. There is a section of this chapter devoted to each of these five steps, and we'll clarify the meaning of each word as we get to it.

The Five Steps of Program Design

Step 1 Make a list of the variables needed to store the numbers you will be using.

Step 2 Write the program lines that will accept user input, the INPUT section of the program.

Step 3 Make a list of the variables that will store the results of the program's calculations.

Step 4 Write the CALCULATION section of the program, the program lines that will compute sales, commissions, and so on.

Step 5 Write the OUTPUT section of the program to produce displays to match your diagram.

In Step 1 you make a list of variables for each of four printers sold, and for the cost of each printer. Variables are a crucial element in programming, so before we can proceed, we must take a very close look at them.

Defining Variables

Variables are tricky. And to keep an explanation of them as simple as possible we'll take an example from the Travel Expense Report program from Chapter 1. Then we'll come back to this chapter's programs.

A variable is like a box that BASIC uses to store numbers. When you use variables in a program, you are, in a sense, preparing empty boxes to be used to store, or hold, values when the program is run. In the Travel Expense Report program, on line 80, we used the variable LX in this INPUT statement:

```
80 INPUT "LODGING EXPENSE"; LX
```

When this line of the program is executed, the screen displays:

```
LODGING EXPENSE?
```

You'll recall that when the BASIC program comes to the end of line 80 it stops everything and waits for you to type the amount of your lodging expense. At this point BASIC has a "box," labeled LX, ready and waiting for the user to enter the lodging expense amount.

When you type in the amount, 200, and press ENTER, BASIC takes the amount you entered, 200, and stores it in the variable "box" labeled LX. The amount then sits there until the rest of the program needs it.

BASIC sets up as many variable "boxes" as it needs to hold the numbers you use in your program. In the Travel Expense program five variables, LX, FX, AX, OX, and TX are used:

Lodging expense	LX
Food expense	FX
Air travel expense	AX
Other expense	OX
Total expense	TX

As the program is run, BASIC fills the variable "boxes" with the numbers the user supplies:

LX, FX, AX, OX, and TX are called the variable names. The meaning of each variable might be clearer if we could label our variable boxes with more descriptive variable names, such as LODGING EXPENSE instead of LX, but Model 100 BASIC only uses two-letter variable names.

Anyway, when you want to know the number that BASIC has in the LX variable box you ask for it by name, the variable name LX. For example, when you type:

```
PRINT LX (ENTER)
```

BASIC looks at all the variable boxes until it finds the one labeled LX. What's in there? The number 200.

The variable "box" is a metaphor that helps us visualize how BASIC stores numbers in variables.

Because variables are so important, there's a lot to learn about them. Variables make it possible to use different data each time the program is run. Thus, once you have a Travel Expense Report program that works, you can use it over and over—all you have to do is enter the amounts you spent each week. The key to good programming is to control the variables and understand what they are doing. The following ideas will help.

Model 100 BASIC only pays attention to *the first two characters* of a variable name. For example, each of these variable names would be recognized by BASIC as unique:

X X2 TX D1 D2 D3 SPEED STORENUMBER

The following variable names would be recognized by BASIC as the *same* variable:

SU1 SU2 SU3 SUB SUN SUM

This is true because Model 100 BASIC only pays attention to the first two characters, SU.

Variable names can't be keywords. Keywords are all the words (PRINT, CLS, INPUT, and so on) that BASIC has reserved for its own use. If you try to use one of these keywords as a variable name, you'll get a syntax error message.

Well-written programs use variable names that remind the programmer of the quantity they symbolize. Using variable names like GT for Grand Total, SM for Sum, TX for tax will make them easier to recall later. If you use cryptic variable names, you'll make it harder to understand your own programs.

Now that you know these rules for creating variable names, you can tackle Step 1 of the program-writing process, making a list of the variables needed to store the numbers the user will later fill in when using the program.

Step 1—Variables for the Input Section

Refer to lines 50–60 of the Printer Sales Report program. For this part of the program we decided we needed eight variables: four for the quantity of each printer sold and four for the current cost of each printer. For quantity we chose the letter Q and identified each type of printer with a number. Thus our variables are Q1, Q2, Q3, and Q4. For current cost we chose the letter C and used the same number per printer (C1, C2, C3, and C4). You will enter these values in the input section, described below.

```
10 'WEEKLY PRINTER SALES REPORT
20 'DATE
30 '
40 '    VARIABLE NAMES -- INPUT SECTION
50 'Q1,Q2,Q3,Q4  QUANTITY OF EACH PRINTER
SOLD
60 'C1,C2,C3,C4  CURRENT SELLING COST OF
EACH PRINTER
70 '    INPUT SECTION
80 CLS
90 PRINT "ENTER QUANTITY OF EACH PRINTER
SOLD"
100 PRINT
110 INPUT "QUANTITY, PRINTER NUMBER 1";Q1
120 INPUT "QUANTITY, PRINTER NUMBER 2";Q2
130 INPUT "QUANTITY, PRINTER NUMBER 3";Q3
140 INPUT "QUANTITY, PRINTER NUMBER 4";Q4
150 CLS
160 PRINT "ENTER CURRENT COST FOR EACH
PRINTER"
170 PRINT
180 INPUT "COST, PRINTER 1";C1
190 INPUT "COST, PRINTER 2";C2
200 INPUT "COST, PRINTER 3";C3
210 INPUT "COST, PRINTER 4";C4
```

Figure 10 The Input Section

Step 2—Input

The input section (lines 80–210) is displayed for the user on two screens, each with its own title. Figure 11 shows how the two screens of the input section appear to the user. Each value entered by the user is stored in the appropriate variable until it is needed.

```
ENTER QUANTITY OF EACH PRINTER SOLD

QUANTITY, PRINTER NUMBER 1? 12
QUANTITY, PRINTER NUMBER 2? 6
QUANTITY, PRINTER NUMBER 3? 3
QUANTITY, PRINTER NUMBER 4? 24
```

```
ENTER CURRENT COST FOR EACH PRINTER

COST, PRINTER 1? 1000
COST, PRINTER 2? 1950
COST, PRINTER 3? 500
COST, PRINTER 4? 300
```

Figure 11 Input Screens

Step 3—Variables for the Calculation Section

In Figure 12, the calculation section of the Printer Sales Report, remarks in lines 230–270 list the variables used in the calculation section. Again, we have chosen variable names that are easily associated with what they represent. These are:

Dollar sales	S1-S4
Commissions	M1-M4
The total weekly quantity	TQ
The total weekly dollar sales	TS
Weekly commissions	TM

```
220 '   VARIABLE NAMES -- CALCULATION AND
OUTPUT SECTION
230 'S1 - S4  DOLLAR SALES OF EACH PRINTER
240 'M1 - M4  COMMISSION ON SALES OF EACH
PRINTER
250 'TQ  WEEKLY TOTAL QUANTITY
260 'TS  WEEKLY TOTAL DOLLAR SALES
270 'TC  WEEKLY TOTAL COMMISSIONS
280 '   CALCULATION SECTION
290 S1=Q1*C1              'DOLLAR SALES
300 S2=Q2*C2
310 S3=Q3*C3
320 S4=Q4*C4
330 M1=S1*.05             'COMMISSION
340 M2=S2*.05
350 M3=S3*.05
360 M4=S4*.05
370 TQ=Q1+Q2+Q3+Q4        'WEEKLY QUANT.
380 TS=S1+S2+S3+S4        'WEEKLY $ SALES
390 TM=M1+M2+M3+M4        'WEEKLY COMM.
```

Figure 12 The Calculation Section

Step 4—The Calculation Section

Once you have defined your variables for the input and the calculation sections the values the user supplies in the input section can be used in the expressions of the calculation section, lines 280–390 in Figure 12. These expressions produce the values for dollar sales, commission, weekly quantity sold, weekly dollar sales, and weekly commission. These new values are again stored by BASIC until they are needed by the output section, described below.

Step 5—The Output Section

As shown in our original diagrams of the output section in Figure 9, the output is divided into two screens. All the values the user supplies and those produced by the calculation section are stored in memory by BASIC. But we must have a way to display them to the user—the output section.

```
PRINTER      QUANT      COST      SALES      COMM
1            Q1         C1        S1         M1
2            Q2         C2        S2         M2
3            Q3         C3        S3         M3
4            Q4         C4        S4         M4

TOTALS       TQ                   TS         TM
```

```
          WEEKLY PRINTER SALES SUMMARY

TOTAL QUANTITY SOLD                    TQ
TOTAL DOLLAR SALES                     TS
TOTAL COMMISSION                       TM
```

Figure 13 Position of Variables in Output Screens

In Figure 13 the position of each variable on the display is shown. Compare Figure 13 with Figure 14, the actual output from the program. The strategy of the output section is simple: Write PRINT statements like lines 420–580 to send each value to the correct location on the screen.

To do this, lines 410–420 in Figure 15 first clear the screen and display the column headings. Then lines 430–480 display the values in their correct positions on the screen. To get the numbers in their correct position BASIC uses a **TAB** function which works much as the TAB on a typewriter does. BASIC's **TAB** allows the programmer to specify how many spaces to the right the **TAB** should move. Here's how the **TAB** function looks:

TAB (*column number*); information

The **TAB** function displays the information following the semicolon at the *column number* specified. BASIC reserves one column for the minus sign, should one be needed. In line 430:

```
430 PRINT "1";TAB(9);Q1;
TAB(17);C1;TAB(24);S1;TAB(32)M1
```

```
PRINTER    QUANT    COST    SALES    COMM
1          12       1000    12000    600
2          6        1950    11700    585
3          3        500     1500     75
4          24       300     7200     360
 ------------------------------------
TOTALS     45                32400    1620
PRESS 'ENTER' TO SEE WEEKLY SUMMARY ...
```

```
          WEEKLY PRINTERS SALES SUMMARY

TOTAL QUANTITY SOLD              45
TOTAL DOLLAR SALES          $32400.00
TOTAL COMMISSION             $1620.00


          PRESS 'ENTER' TO CONTINUE . . .
```

Figure 14 Output Screens from Sales Report Program

```
400 '    OUTPUT SECTION
410 CLS
420 PRINT
"PRINTER  QUANT  COST  SALES  COMM"
430 PRINT
"1";TAB(9);Q1;TAB(17);C1;TAB(24);S1;
TAB(32); M1
440 PRINT
"2";TAB(9);Q2;TAB(17);C2;TAB(24);S2;
```

Figure 15 continues

```
TAB(32); M2
450 PRINT
"3";TAB(9);Q3;TAB(17);C3;TAB(24);S3;
TAB(32); M3
460 PRINT
"4";TAB(9);Q4;TAB(17);C4;TAB(24);S4;
TAB(32); M4
470 PRINT "-----------------------------
----------"
480 PRINT "TOTALS";TAB(9);TQ;TAB(24);
TS;TAB(32);TM
490 LINE INPUT "PRESS 'ENTER' TO SEE WEEKLY
SUMMARY...";X$
500 '  WEEKLY SUMMARY
510 CLS
520 PRINT "      WEEKLY PRINTERS SALES
SUMMARY"
530 PRINT
540 PRINT "TOTAL QUANTITY SOLD";TAB(28);
TQ
550 PRINT "TOTAL DOLLAR SALES";TAB(23);
560 PRINT USING "$$######.##";TS
570 PRINT "TOTAL COMMISSION";TAB(23);
580 PRINT USING "$$######.##";TM
590 PRINT
600 PRINT
610 LINE INPUT "    PRESS 'ENTER' TO
CONTINUE...";X$
620 CLS
630 PRINT "      ENTER NUMBER OF YOUR
CHOICE"
640 PRINT
650 PRINT "      1-REVIEW PREVIOUS RESULTS"
660 PRINT "      2-ENTER NEW DATA"
670 PRINT "      3-END PROGRAM"
680 PRINT:PRINT
690 INPUT "      THEN PRESS 'ENTER'";X
```

Figure 15 continues

```
700 ON X GOTO 400,80,710
710 CLS:END
```

Figure 15 Printer Sales Report Program: Output Section

the value of Q1 will be displayed beginning in the ninth column of the screen. Similarly the value of C1 will be displayed in the 17th column, S1 in the 24th, and M1 in the 32nd column. Lines 430–480 repeat this format, aligning the numbers in orderly columns and rows.

Line 490 uses the LINE INPUT statement in a special way. If line 490 is omitted, line 510 is executed immediately and the screen clears before the user has a chance to read the output. Normally BASIC waits at the end of the LINE INPUT statement for the user to enter data. In line 490, the program simply waits for the user to press ENTER. We're using the LINE INPUT statement to make BASIC wait while we read the screen. The variable X$ must be included at the end of line 490 or this trick won't work.

Lines 510–610 in Figure 15 cause a summary of the week's sales activity to be shown on the screen (see the second screen in Figure 14). PRINT USING statements provide the dollars-and-cents format for weekly total dollar sales and commissions. Line 610 serves the same purpose as line 490, freezing the display until the user types ENTER.

The five fundamental steps of the program-writing process have now been completed. Other program-writing strategies will be described in later chapters.

A Menu

Lines 620–710 shown in Figure 16 create a menu. A menu offers the user a list of options that he or she may wish to do next. In this program the user is offered three choices as shown below in Figure 16.

In line 690 an INPUT statement allows the user to enter a 1, 2, or 3, which is stored in the variable X.

Then in line 700 the program uses a new statement, an **ON . . . GOTO** statement, to see which number was entered. The **ON . . . GOTO** statement follows the form:

ON *X* **GOTO** *list of numbers*

where the value of X determines the line number to which the program will jump. In line 700:

```
700 ON X GOTO 400,80,710
```

when X (from line 690) equals 1 the program jumps to line 400, and the results of the program are displayed for review by the user. When X=2, the program jumps to line 80, and the user can start the program over again, entering new data. When X=3, the program ENDS.

```
ENTER NUMBER OF YOUR CHOICE

1-REVIEW PREVIOUS RESULTS
2-ENTER NEW DATA
3-END PROGRAM


THEN PRESS 'ENTER'?
```

Figure 16 Menu Screen from Sales Report Program

Menus can be used anywhere in a program to provide user control. Menus are a user-friendly device because they lay the choices out in front of the user and give the user a simple way of designating the path he or she wants to take.

If you are a new programmer the first half of this chapter contained a lot of information new to you. You have really done well to have come this far. We still have a lot more to say about variables, so this is a good spot to quit if you'd like a break.

More About Variables

Model 100 BASIC provides you with four variable types. Variable types refer to the different kinds of variables in BASIC. There are only four kinds and they have the following very jargony names:

double precision variable

single precison variable

integer variable

string variable

Each of these variables is identifiable by its type declaration tag.

Type declaration tags are like baseball uniforms. You can tell what team the players are on if you look at the uniforms. You can also tell what type of variable is being used by looking at the type declaration tag. These tags are placed as suffixes after the last character of a variable name to identify the variable type:

Type Declaration Tag	**Variable Type**
# (or no tag)	double precision
!	single precision
%	integer
$	string

For example:

X#	is a double precision variable.
X	is also a double precision variable, because a variable without a type declaration tag is double precision as far as BASIC is concerned.
T!	is a single precision variable.
NR%	is an integer variable.
WD$	is a string variable.

When you type in your own programs you will probably only use double precision and string variables. However, single precision and integer variables are included in this section so that if you see them used in other programs you will know what they are. We'll look at these types of variables one at a time below.

Double Precision Variables

In the Model 100, double precision variables store numbers of up to 14 digits, such as:

98765432101234

This means that if you have more than $999999999999.99 dollars (with 14 nines and *no* commas!) in a checking account, the Model 100 can't balance your checkbook. Perhaps you should split the money between two accounts.

Model 100 BASIC treats all variables as double precision unless you specify otherwise. This means, for all practical purposes, that every variable you write in a program can store a number 14 digits long. Therefore don't use the other variable types if you want to calculate the United States deficit on the Model 100.

Single Precision Variables

Single precision variables store numbers with up to six digits. This is a number with 6 digits:

123456

This means you probably won't use a single precision variable to store your gross annual income since $9999.99, with 6 digits, is the largest dollar value that a single precision variable can store.

Integer Variables

Integer variables store whole numbers. A whole number is a number without a decimal point, like this:

24516

Model 100 BASIC integer variables can store whole numbers between −32768 and 32767. What does this mean in the practical world of business? You can't use integer variables to hold the zip codes in your mailing list program. True, zip codes don't have decimal points, *but* they do go up to about 97777, while integers in BASIC only go to 32767.

Strings and String Handling

We said in Chapter 1 that string variables store words as opposed to numerical values.

These variables are called strings because they store strings of characters. From a computer's point of view, a word can be composed of any letters, numerals, or symbols.

Strings have many distinguishing qualities. For instance a string variable can store any string of up to 255 characters. All the following strings are valid:

```
CR$  =  "Radio Shack TRS-80, Model 100"
PB$  =  "Books, Inc."
TL$  =  "8005551234"
AD$  =  "Jones:85 Center Street:New York, NY"
PH$  =  "5551234(?pA?PDOWl·;?WDJNS M?
        password M)"
M$   =  "OINGO-BOINGO"
```

Strings do not even have to make sense because they are stored literally, exactly as they are input. And strings can be used over and over, just like other variables. Part of the weekly Printer Sales Report program could have been written as follows:

```
30 HD$ = "PRINTER  QUANT  COST
SALES  COMM"
.
.
.
400 'OUTPUT SECTION
410 CLS
420 PRINT HD$

PRINTER   QUANT   COST   SALES   COMM
```

The string HD$ could be used again and again as column headings. In a larger program with multiple screens HD$ could save typing

and memory space, an important consideration if you have the 8K version of the Model 100.

Another quality that distinguishes strings is their ability to undergo incredible transformations. A string is a set of discrete recognizable characters. Therefore, you can dissect a string to select any subset of those characters. For example, in the string "OINGO-BOINGO" a subset could be "BOINGO." New strings may also be constructed from old, as follows:

```
10 A$ = "THIS"
20 B$ = "AND THAT"
30 C$ = A$ + B$
40 PRINT C$

THIS AND THAT
```

There are special string functions that facilitate the transformation process. The string functions are listed in the Model 100 owner's manual, but some of them are also described later in this chapter.

String Variables and the ASCII Code

It may surprise you to learn that before a computer can use a string, the letters in it are converted to numbers. Computers always store information as numbers because computers only understand numbers. So the people who invented computers came up with a system that assigns a specific number to each letter, numeral, and symbol used by the computer. When the computer wants to store a letter, such as the letter T, it stores its code number, 116. The codes are called ASCII.

ASCII stands for American Standard Code for Information Interchange. ASCII characters are a standard way of identifying letters, numbers, and symbols. This information is sometimes called alphanumeric information.

Look at the following program:

```
10 W$="TELEVISION"
20 PRINT W$

TELEVISION
```

Here the string variable W$ stores a string of ASCII characters corresponding to the letters T-E-L-E-V-I-S-I-O-N. The *ASCII value* of each character in TELEVISION is:

Character	ASCII Value
T	116
E	101
L	108
E	101
V	118
I	105
S	115
I	105
O	111
N	110

The string TELEVISION is stored in memory as these ten ASCII values.

DAY$, DATE$, and TIME$ Functions

The Model 100 has built-in string functions, DAY$, DATE$, and TIME$. You set the DAY$, DATE$, and TIME$ in Chapter 0. The sample program below illustrates that these functions can be used in any BASIC program to display the day of the week, the date, or the time of day. Enter this program in your Model 100 and see for yourself.

```
10 PRINT DAY$
20 PRINT DATE$
30 PRINT TIME$

Tue
08/30/83
16:03:31
```

The DAY$, DATE$, and TIME$ values are provided by the Model 100 as strings. TIME$ is pronounced TIME; the dollar sign

is not pronounced as an "S". The dollar sign is the type declaration tag (the baseball uniform) and indicates that TIME$ is a string variable.

You may remember from Chapter 0 that the TIME$ function gives the time, including the seconds, in 24-hour military notation. For example, 9:05 A.M. is displayed as 09:05:00. But 9:05 P.M. is displayed as 21:05:00. If you find the Model 100's 24-hour time distracting, you might enjoy using the Twenty-Four Hour Clock Conversion Program (see Figure 17). This program translates 24-hour notation into "A.M." and "P.M." time. It uses string functions to transform the 24-hour time supplied by Model 100 BASIC into normal A.M. or P.M. time. It uses string functions to transform the 24-hour time supplied by Model 100 BASIC into normal A.M. or P.M. time, without the seconds. The program takes advantage of a few unique features of the Model 100 and illustrates the use of string functions in a practical application.

```
10 'TWENTY FOUR HOUR CLOCK CONVERSION
PROGRAM
20 CLS:PRINT:PRINT TAB(9)
30 HOUR$=LEFT$(TIME$,2)
40 T=VAL(HOUR$)
50 IF T=12 THEN GOTO 110
60 IF T>12 THEN GOTO 100
70 '   PRINT TIME$ IF A.M.
80 PRINT "The time is ";LEFT$(TIME$,5)
;" a.m.":GOTO 150;
90 ' CONVERT TO P.M. "REGULAR" TIME
100 T=T-12
110 T$=STR$(T)
120 PM$=T$+MID$(TIME$,3,3)
130 ' PRINT PM$ AS P.M. TIME
140 PRINT "The time is ";PM$;" p.m."
150 PRINT
160 PRINT TAB(9);DAY$;"          ";DATE$
170 PRINT
180 PRINT "   PRESS 'F4' FOR CURRENT TIME
..."
```

Figure 17 Clock Conversion Program

The Clock Conversion Program

Line 30 uses the LEFT$ function to begin the conversion by extracting "hours" from the TIME$ string. The LEFT$ function takes the form:

LEFT$ (*string, number of characters*)

Line 30 works by extracting the *number of characters* (2) from the LEFT of the string. Thus, the first two characters from the left of "21:05:00" are extracted.

BASIC provides the VAL function, used in line 40, to convert the HOUR$ string "21" to the numeric value, T=21.

Lines 50 and 60 use the numeric value of T in IF . . . THEN statements to distinguish between A.M. and P.M. time.

```
50 IF T=12 THEN GOTO 110
60 IF T>12 THEN GOTO 100
```

IF the value of T equals 12 THEN control is passed to line 110. IF (as in our example) the value of T is greater than 12 THEN control is passed to line 100. IF the value of T is less than 12 (A.M.) THEN control falls through to line 80. Please note that Lines 70 and 90 are remark lines.

Let's see what happens to our example when the program jumps to line 100.

Line 100 converts 24-hour P.M. time to regular P.M. time by subtracting 12 from the hours. Then in line 110 the STR$ function converts the numeric value of T, 9, to the string, T$, "9".

Now put T$ together with the minutes portion of TIME$ and we have regular P.M. time. This is done in line 130 with the MID$ function. It takes the form:

MID$ (*string, position, number of characters*)

MID$ works by extracting the first *number of characters* from the MIDdle of the string, beginning at the *position* in the string indicated. Line 120 uses MID$ in the expression

```
120 PM$=T$+MID$(TIME$,3,3)
```

to select three characters from the string 21:05:00, beginning with the third character. We use the number 3 in the MID$ function here because we want the minutes portion of the TIME$ string, beginning with the third character of the string. The result is a string, the minutes portion of TIME$, ":05".

Remember that we can add strings. We saw the sample program,

```
10 A$ = "This "
20 B$ = "and that."
30 C$ = A$ + B$
40 PRINT C$

This and that.
```

Well, line 120 also adds two strings. The string T$ is added to the string MID$ (TIME$,3,3). The result is PM$. Here's line 120:

```
120 PM$=T$+MID$(TIME$,3,3)
```

Here's how it puts these two strings together:

```
T$="9"
MID$(TIME$,3,3)=":05"
```

Therefore, the two strings, "9" and ":05" become "9:05." Line 140 displays the result.

```
                The time is 9:05 p.m.
                  Tue      08/30/83
           PRESS 'F4' FOR CURRENT TIME ...
 Ok
```

Figure 18 Clock Conversion Program Output

Figure 18 shows the display produced by the program. Line 160 also displays the DAY$ and DATE$ for your convenience.

Line 180 will send a message to the screen instructing the user to press one of the Model 100's eight function keys. F4 is the RUN key. By simply pressing key F4, the user can reRUN the program to display the current time.

More About Strings

Strings can also be used to write a mailing list program for your Model 100. You could put all your friends on your mailing list and send them letters that say, "You might have already won $1,000,000.00." Or you could put all your customers on your mailing list and send them mailings to promote your business. A mailing list can also be used as a customer list—a kind of business address book.

Because mailing list programs are used every day in business we're going to develop one for the Model 100 throughout the next few chapters. We can't do it all at once. We'll start by telling how string variables are used in the mailing list program.

String variables, of course, are used in the mailing list program because they store words. Figure 19 contains the input section from the customer mailing list program. It accepts customer name and address information from the keyboard. Then it displays the information on the screen for you to review. Parts of programs that accomplish limited tasks are sometimes called routines. The lines of code in Figure 19 illustrate the beginning of our mailing list program, which is called a data entry routine.

```
10 'DATA ENTRY ROUTINE
20 'DATE
30 '
40 '  INPUT VARIABLES
50  'CN$ COMPANY NAME
60  'ST$ STREET AND NUMBER
70  'CS$ CITY STATE
80  'PH$ PHONE
90  'NM$ NAME OF PERSON TO CONTACT
```

Figure 19 continues

```
100 'PR$ PRODUCT CODE
110 '   DEFINE HEADINGS
120 C$="CO. NAME......."
130 S$="STREET ADRS...."
140 T$="CITY/STATE....."
150 P$="PHONE.........."
160 N$="NAME..........."
170 R$="PRDCT CODE....."
180 'INPUT SECTION
190 CLS
200 PRINT "TYPE INFO REQUESTED, THEN
'ENTER'..."
210 LINE INPUT "CO. NAME -- ";CN$
220 LINE INPUT "STREET & NUMBER -- ";ST$
230 LINE INPUT "CITY & STATE -- ";CS$
240 LINE INPUT "PHONE NUMBER -- ";PH$
250 LINE INPUT "PERSON TO CONTACT - - ";NM$
260 LINE INPUT "PRODUCT CODE 1,2,3 OR 4 --
";PR$
500 '  DISPLAY DATA ON SCREEN FOR REVIEW
510 CLS
520 PRINT C$;CN$
530 PRINT S$;ST$
540 PRINT T$;CS$
550 PRINT P$;PH$
560 PRINT N$;NM$
570 PRINT R$;PR$
580 PRINT
590 LINE INPUT "PRESS 'ENTER' TO CONTINUE
...";X$
600 GOTO 180
```

Figure 19 Data Entry Routine

Remark lines 50–100 list the string variables that will store the input from the keyboard. Then six assignment statements, lines 120–170, define the headings to be used in the output section. These lines just set things up for use later in the program.

The input section, lines 190–260, accepts the customer in-

formation via LINE INPUT statements. The information is then displayed on the screen in lines 520–570 for the user to review. Notice the structure of the PRINT statements, for example:

```
520 PRINT C$;CN$
```

With only five characters and a semicolon, line 520's PRINT statement sends the heading *and* the company name information to the screen as shown on the second screen in Figure 20. The twelve variable names, in lines 520–570, fill the screen with information. This is one power of string variables in business data programs. Remember that a semicolon between two variables causes them to be PRINTed one after the other on the same line.

When the mailing list program is complete you'll be able to revise the data, or send the data to a file for storage. These steps will be added when file handling is covered in Chapter 5.

```
TYPE INFO REQUESTED, THEN 'ENTER'...
CO. NAME -- QWERTY INTERNATIONAL,LTD.
STREET & NUMBER -- 26 ALPHA WAY
CITY & STATE -- WRIGHT, OK
PHONE NUMBER -- 8005551212
PERSON TO CONTACT -- MR. JOHN QWERTY
PRODUCT CODE 1,2,3 OR 4 -- 3
```

```
CO. NAME.......QWERTY INTERNATIONAL,LTD.
STREET ADRS....26 ALPHA WAY
CITY/STATE.....WRIGHT, OK
PHONE..........8005551212
NAME...........MR. JOHN QWERTY
PRDCT CODE.....3
PRESS 'ENTER' TO CONTINUE...
```

Figure 20 Output Screen

CHR$ Function

Up until now BASIC has been converting the letters that we type to ASCII. There is more to ASCII characters than just the ABC's. There are times, thank goodness not very often, when we have to tell the computer exactly what ASCII code we need. Here's why.

In addition to ASCII codes for letters of the alphabet and symbols like @, the ASCII character chart also contains ASCII codes to control certain functions on the Model 100. For example, when the Model 100 receives the ASCII character 7, it sounds the buzzer. When it receives the ASCII character 12, it clears the screen.

There are even ASCII codes for text formatting such as space (ASCII 32), tab (ASCII 9), and carriage return (ASCII 13). Its a good thing that we don't have to remember all these codes most of the time. But when you need them the *CHR$* function gives you a way to send these codes to the Model 100 from within a BASIC statement. The CHR$ function takes the form:

PRINT CHR$ (*decimal value of ASCII code*)

Remember, the CHR$ function is a string function. It sends a string from the BASIC program to the Model 100. Even though there are commands in Model 100 BASIC to sound the buzzer (BEEP) and clear the screen (CLS), the CHR$ command is needed at times, such as when programming the function keys.

Programmable Function Keys

Remember the function keys described in Chapter 0? By typing a single function key you could give the Model 100 a command—LOAD, SAVE, RUN, LIST, or MENU.

You can program the function keys to do exactly what you want by changing their function with KEY. KEY is a string function unique to the Model 100. KEY allows you to assign new functions to the function keys, F1–F8. KEY assigns a string to the function key. When you press the designated function key it sends that string to the Model 100.

We mentioned that key F4 RUNs a program, any program. The string associated with the F4 key is "RUN" + CHR$(13).

CHR$(13) stands for carriage return or ENTER. When the Model 100 receives the string CHR$(13) it reacts just as if you pressed ENTER. So when you press F4 it is as if you typed RUN and then pressed ENTER. To program a function key use the expression,

KEY *function key number, string expression*

The *string expression* can only contain 15 characters. You can program F6 by typing,

KEY 6, "EDIT"+CHR$(13)

Then pressing F6 will put you in the TEXT program editing mode.

The function keys can be programmed to save you time when writing programs, too. If you are writing a program with many LINE INPUT statements, you can program F7, for example, by typing:

KEY 7, "LINE INPUT "+CHR$(34)

Then press ENTER. 34 is ASCII code for quotation marks. Pressing F7 will automatically write:

```
LINE INPUT "
```

Then you can complete typing the statement. If you do this you will never again have to type LINE INPUT ".

The next chapter shows you more ways to organize a BASIC program to solve a problem. Branching and conditional statements, covered in detail, will show you how to control your programs so that they follow the problem-solving method that you devise.

Chapter 3

Going with the Program Flow: Branching, Conditional Statements, and Loops

In the last chapter you started to look at the ways that BASIC programs are organized. You also saw how BASIC uses variables to store and manipulate information. In this chapter you'll dig even deeper into the program-writing process. And you'll learn some very important commands that will enable you to control your BASIC programs and solve problems.

This chapter contains many sections. It's sort of a "Mini-Handbook of BASIC Problem Solving Commands." Like any handbook, it's filled with examples of how to use the commands. Take this chapter at your own pace, but don't skip it. You've come this far and deserve the best kind of attention.

This chapter teaches you a very important lesson, so important that it's:

THE FIRST RULE OF PROGRAMMING:

When you want to write a BASIC program to solve a problem, first organize the problem so that it can be solved by a BASIC program.

You'll learn to duplicate the steps you might take with pencil and paper by organizing them into a series of BASIC instructions. That brings us right away to:

THE SECOND RULE OF PROGRAMMING:

If a task can't be understood and completed manually, it probably can't be programmed successfully.

The computer program may improve on the manual system. It may be faster, more accurate, easier to use, and require less work. But in most cases a program won't do anything that the manual system didn't do. This means that to write a program you have to understand the steps needed to complete the task you are programming. And true to form, computer people have a gigantic word for the steps need to complete a task—the algorithm.

The Algorithm

An algorithm is any finite series of steps that results in the solution to a problem. Just to take the technical edge off the word, here's an algorithm for drinking a cup of coffee:

1 Pick up the cup.

2 Raise it to your mouth.

3 Take a sip...
...don't spill any on your shirt.

4 Put down the cup.

5 Repeat steps 1–4 until cup is empty or until you're called out of the office for a meeting.

An algorithm can be simple or complicated. It depends on what the algorithm is for. The coffee drinking algorithm is fairly simple, whereas an algorithm for launching the Space Shuttle is very complicated.

In programming, the word *algorithm* refers specifically to the program steps you formulate to solve a problem.

After this big buildup, you may be disappointed to learn that we won't explain how to create an algorithm to follow the problem-solving logic of business problems until the second part of this chapter. First we need to take a look at the problem-solving tools that BASIC provides. Without these tools it is not possible to create an algorithm.

Branching, Conditional Statements, and Loops

As we saw in Chapter 1, programs follow a series of instructions until they achieve a desired result. The program examples in Chapters 1 and 2 started at the first BASIC line and proceeded to the END statement, following each BASIC instruction in numerical order. When some programs got to the end you had an option to start over again, but other than that they followed a straight line.

You've probably never solved a problem this way, because real-life problem solving requires a more complex, roundabout approach. We make comparisons. We make logical choices. We take one path instead of another. We fall back on well-known methods in the middle of a new problem. We follow one route if certain conditions are met, but we take a completely different course if other conditions prevail. And we retrace our steps as we perform repetitive tasks.

BASIC can pretty much duplicate all of these human problem-solving techniques by using branching, conditional statements, and loops.

There's a separate section of this chapter for each of these problem solving tools and we'll take them slowly. So first let's go to GOTO.

GOTO

The **GOTO** statement is called an unconditional branching statement. It follows the form:

GOTO *line number*

where the *line number* is the number of a line in the BASIC program, such as:

```
GOTO 20
```

The **GOTO** statement sends the program backward or forward to whatever line number is specified. **GOTO** 20 would send the program to line 20. It's up to the programmer to decide what to write on line 20.

When the program takes a side trip like this it is called branching because, like the branch line of a railroad, the program forms a loop off of the main line. A loop is a section of code that is executed more than once during one complete pass through a program (there is more about loops later).

GOTO is called an unconditional branching statement because the **GOTO** is executed no matter what other conditions prevail. That means that *whenever* the program comes up to the **GOTO** statement the program branches to the line indicated, no matter what. Branching is sometimes called jumping because the program jumps over one or more lines of the program to reach the line indicated by the **GOTO** statement. The program below illustrates the use of **GOTO:**

```
10 X=1
20 PRINT X;
30 X=X+1
40 GOTO 20
50 END
```

In this little program, line 10 sets the variable X equal to 1. Line 20 displays the value of X on the screen. Line 30 adds 1 to the value of X. Then line 40 uses the **GOTO** statement to jump to

line 20 and start the process over again. Enter this program on the Model 100 and RUN it.

The screen starts off displaying 1. Then it displays 2 . . . then 3 . . . then 4 . . . and so on *forever*. Don't worry, your computer isn't broken. This is called an endless loop. It's included here to show you what looping is all about. Of course it doesn't do anything except demonstrate that when the program gets to line 40, an unconditional branching statement, it always jumps back to line 20. You can stop execution of this loop by pressing the SHIFT key and the BREAK command key at the same time. (This loop won't really go on endlessly. It will continue until X is too big to be calculated by BASIC—1 followed by 63 zeros.)

GOTO statements can be used anywhere in a program, but obviously there must be a good reason to do so. Here's how to fix the sample loop program so that it only displays the numbers 1 through 10:

```
10 X=1
20 PRINT X;
30 X=X+1
35 IF X = 10 THEN GOTO 50
'new program line
40 GOTO 20
50 END
```

Remember the IF . . . THEN statement from the TRAVEL program in Chapter 1? Line 35 uses an IF..THEN statement to keep the loop from continuing endlessly. Each time the program gets to line 35, the IF..THEN statement compares the value of X to 10. If X is not yet equal to 10 the program ignores line 35 and continues on to line 40. The **GOTO** statement in line 40 sends the program back to line 20.

But when X = 10 the IF . . . THEN statement gets the attention of the program, so to speak. **GOTO** 50, to the right of THEN, sends the program to line 50, where the program ends. This prevents the loop from going on forever.

GOSUB

We mentioned earlier in this chapter that humans fall back again and again on well-known methods in the middle of solving a big problem. Right in the middle of trying to make a sale to a new customer, for instance, you might fall back on a sales trick that you've used dozens of times.

Programs, too, often contain a few program lines that the program will use again and again. BASIC gives us a place to put these lines, and that place is called a subroutine. A subroutine is a section of code, a few program lines that perform a function that can be used by the program whenever that function is required. Subroutines can contain instructions for calculations, input, output, or any other task required by more than one section of the program.

The **GOSUB** (**GO** to **SUB**routine) statement sends the program to a subroutine. Using a **GOSUB** statement is sometimes referred to as calling a subroutine. The **GOSUB** statement follows the form:

GOSUB *Line Number*

The *line number* is the number of the first line of the subroutine. The last instruction in the subroutine is always RETURN.

The RETURN sends the interpreter to the line following the **GOSUB** statement so that the execution of the main program can be resumed.

The program shown in Figure 21 and the output shown in Figure 22 illustrate the **GOSUB** statement and how a subroutine creates the output section of the program.

We'll go through the program slowly, one line at a time, because we'll also review how strings are used by the program. Please note that this program doesn't *do* anything except demonstrate a subroutine and the use of strings. As the book progresses you'll see that these commands are used often, especially in programs that ask for names, such as mailing list programs.

Sending the program to a subroutine is like going on a trip. When you go on a trip you have to make sure that you have everything you need before you leave. When you send a program to a subroutine you have to make sure that the program has everything

it will need in the subroutine before it gets to the **GOSUB** statement.

When the program gets to the subroutine in lines 140–150 the values of the variables B$ and A$ will have to have been supplied by the user. So before we send the program to the subroutine we have to assign values to B$ and A$. LINE INPUT statements in lines 40–50 get the values of the variables B$ and A$ from you.

```
10 '  PROGRAM USING SUBROUTINE TO
20 '  SEND INFORMATION TO THE SCREEN
30 CLS
40 LINE INPUT "ENTER COMPANY NAME...";A$
50 B$="COMPANY "
60 GOSUB 130
70 LINE INPUT "ENTER PRODUCT NAME...";A$
80 B$="PRODUCT "
90 GOSUB 130
100 PRINT
110 LINE INPUT "PRESS 'ENTER' TO CONTINUE
...";X$
120 GOTO 30
130 '  SUBROUTINE ACTS AS OUTPUT SECTION
140 PRINT B$;"NAME IS ";A$
150 RETURN
```

Figure 21 Program with Subroutine

Line 40 is a LINE INPUT statement. LINE INPUT asks you to:

```
ENTER COMPANY NAME . . .
```

You type:

INTERCORP, INC. (ENTER).

The string variable, A$, at the end of line 40 represents INTERCORP, INC. That takes care of A$.

Then line 50 assigns the word COMPANY to the string variable, B$. So at this point, just before the program gets to line 60, A$ = INTERCORP, INC., B$ = COMPANY.

Line 60 then sends the program to the subroutine at line 130 with the **GOSUB** statement:

```
60 GOSUB 130
```

This particular subroutine is just a PRINT statement, and you may feel like a PRINT statement expert by now! Line 140 sends the line, COMPANY NAME IS INTERCORP, INC., to the screen, as shown in the second line of Figure 22.

```
ENTER COMPANY NAME...INTERCORP, INC.
COMPANY NAME IS INTERCORP, INC.
ENTER PRODUCT NAME...A100, AMPS
PRODUCT NAME IS A100, AMPS

PRESS 'ENTER' TO CONTINUE...
```

Figure 22 Output Screen from Previous Program

The RETURN in line 150 RETURNs program control to line 70, (the line *following* the **GOSUB** statement).

Line 70 is a LINE INPUT statement which requests the product name and again uses the variable A$. The user types in:

A100, AMPS (ENTER)

Line 80 assigns the string PRODUCT to variable B$. So at this point, just before the program gets to line 90, and the program's second **GOSUB,** A$ = A100, AMPS, B$ = PRODUCT.

You may have noticed that the program uses the variable A$ and B$ again. Variables can be used again and again, as long as it doesn't matter that their previous values are lost. More than one part of a program can use the same subroutine, as long as each one sends the subroutine the right values.

So, ready with the values for A$ and B$, line 90 calls the

subroutine again and the output process is repeated.

Whenever you send the program to a subroutine make sure that the last line of the subroutine is a RETURN and that the program gets to the RETURN. This is:

> THE THIRD RULE OF PROGRAMMING:
>
> Always enter a subroutine using a **GOSUB** statement, never a GOTO, because only the **GOSUB** tells the subroutine where you came from (so the RETURN knows where to return you to). Also, always leave the subroutine with RETURN, never a GOTO.

The second half of THE THIRD RULE needs an explanation. You can't just jump out of the subroutine with a GOTO, because BASIC is waiting to use the RETURN address when you're done with the subroutine. BASIC stores these RETURN addresses in memory. If you leave a huge pile of unused RETURN addresses the computer literally runs out of room to store them and your program will crash with an ?OM Error.

That ?OM error message means that BASIC is Out of Memory. So avoid breaking THE THIRD RULE.

IF . . . THEN . . . ELSE Statements

We had one look at IF . . . THEN statements in Chapter 1. We'll give them a quick review and then take a look at a new addition—ELSE.

But first, remember that in real-life problem solving we do one thing if one condition exists, but we do something else if there is any other condition.

For example, let's say you have to decide what to eat for lunch. If you have less than $5 you have pizza, but if you have more than $5 you have steak. This type of decision is called a conditional decision because the answer depends on the condition (of your wallet). We can make these conditional decisions in BASIC with conditional branching statements, like the *IF . . . THEN . . .* ***ELSE*** statement. The IF . . . THEN . . . **ELSE** statement follows the form:

IF *(this expression is true)* THEN *(execute this first command)* **ELSE** *(execute this second command)*

Let's use our lunch problem to develop a small sample program. The heart of this sample program is an IF . . . THEN . . . **ELSE** statement. It works this way:

IF *(you have less than $5)* THEN *(eat pizza)* **ELSE** *(eat steak)*

This program answers the question, "What should I have for lunch today?" in the following way:

```
10 INPUT "ENTER CASH IN WALLET . . .
"; CW
20 IF CW<5 THEN GOTO 30 ELSE GOTO 50
30 PRINT "EAT PIZZA"
40 END
50 PRINT "EAT STEAK"
60 END
```

Type in this program and try it. Line 10 starts off with an INPUT statement, and asks you to:

ENTER CASH IN WALLET . . .

If you have $4, you type:

ENTER CASH IN WALLET . . . 4 (ENTER)

The amount that you enter (4) is represented by the variable CW (CW is the variable in Line 10; it stands for Cash in Wallet).

The program proceeds to the IF . . . THEN . . . **ELSE** statement in line 20. The first part of the IF . . . THEN . . . **ELSE** statement looks like this:

```
20 IF CW<5 THEN GOTO 30
```

The symbol (<) means *less than*. BASIC is saying: IF CW is less than 5 THEN go to line 30. Since CW is 4, and 4 is less than 5, the program jumps to line 30. Line 30 is just a PRINT statement. It displays the answer to your problem,

```
EAT PIZZA
```

OK. But what if it's Friday? You just got paid and you cash a check for $100. Run the program again. Now when the INPUT statement in line 10 asks for your cash position you answer by typing 100 (ENTER):

ENTER CASH IN WALLET . . . 100 (ENTER)

And once again the program proceeds to the IF . . . THEN . . . **ELSE** statement in line 20. But this time the program goes to the **ELSE** side of line 20:

```
20 IF CW < THEN GOTO 30 ELSE GOTO 50
```

Now CW = 100. 100 is definitely *not* less than 5. So the program is sent to line 50. Line 50 displays the answer. This time you find it more palatable:

```
EAT STEAK
```

By the way, in an IF . . . THEN . . . **ELSE** statement, the **ELSE** portion of the statement may be omitted. If the **ELSE** is omitted and the first expression is not true, control falls through to the next line in the program. That's why this book refers to them as IF . . . THEN statements whether they have the **ELSE** or not.

Without using **ELSE** our lunch program could have been written as follows:

```
10 INPUT "ENTER CASH IN WALLET . . . "; CW
20 IF CW<5 THEN GOTO 50
30 PRINT "EAT STEAK"
40 END
50 PRINT "EAT PIZZA"
60 END
```

In this modified program, if you have less than $5 the program goes to line 50 and you EAT PIZZA. If you have more than $5 the

program falls through to line 30 and you EAT STEAK.

How do you, as a programmer, know when to use the IF . . . THEN . . . **ELSE** statement? Simple. Whenever you hear yourself asking a conditional question.

We know that you're not going to write a program to decide what to have for lunch. But the IF . . . THEN . . . **ELSE** decision-making techniques in the lunch program can be used to make serious decisions. Here are a couple that will help give you a sense of how to use IF . . . THEN . . . **ELSE:**

IF *(bank statement = my checkbook balance)* THEN *(everything's OK)* ELSE *(look for error)*

IF (mortgage payments are less than $800) THEN *(get mortgage)* ELSE *(wait for interest rate to drop)*

More About IF . . . THEN Statements

If you've had enough of IF . . . THEN statements, THEN you are probably human, ELSE you may be part computer. But because IF . . . THEN statements lie at the heart of every decision you can make with your computer there's a lot to learn about them.

Therefore, IF you want to learn all there is to know about IF . . . THEN statements THEN read on, ELSE skim this section. The next few pages are examples of the dozens of ways that an IF . . . THEN statement is used.

Conditional statements are used in two ways. There are conditional branching statements and conditional statements that affect the value of variables.

Here are some examples of conditional branching statements:

```
IF X=10 THEN GOTO 150
```

If X=10 then the program jumps to line 150. Any other value of X allows control to fall through to the next line.

```
IF A=3 THEN GOSUB 500 ELSE GOTO 1000
```

If A=3 then the subroutine beginning at line 500 is called. Any other value of A sends the program to line 1000.

```
IF X$="Y" THEN GOTO 10 ELSE END
```

If the string variable X$ equals "Y" then the program proceeds to line 10. Any other value of X$ causes the program to end. (This could be used when you want to ask the question, "Would you like to run the program again?")

Here are some examples of conditional statements that change the value of variables:

```
IF Z=100 THEN Y=0
```

If the value of Z equals 100 then the value 0 is assigned to Y. Any other value of Z allows control to fall through to the next line and the value of Y will be unaffected.

```
IF A=12 THEN A=0 ELSE A=A+1
```

If the value of A has reached 12 then reset the value of A to zero. If A has not reached 12 add 1 to A. (This could be used when you want the program to go through a loop 12 times and then start again and go another 12 times, and so on for each month of the year.) For example:

```
10 A=0
20 PRINT A
30 IF A=12 THEN A=0 ELSE A=A+1
40 GOTO 20
```

```
IF X$="Y" THEN A=0 ELSE A=A+1
```

If you answer "Y" to a line input then A equals zero. If not, 1 is added to A.

In all these examples the IF . . . THEN statement uses one or more expressions between the IF and the THEN to see if the value of the variable is *equal* to some other value. But IF . . . THEN statements can also check for *inequality*. The following symbols are used for this purpose:

Symbol	Meaning
=	equal to
< >	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

For example, this conditional statement checks for inequality:

```
IF X<100 THEN GOTO 10 ELSE X=X-1
```

If X is less than 100 THEN the program goes to line 10. Any value of X that is 100 or greater causes 1 to be subtracted from X.

When writing IF . . . THEN statements it's usually better *not* to check only for equality. Let's say you want to subtract 1 from X until X equals zero and then end the program. You could use these program lines:

```
10 X=200
20 PRINT X
30 X=X-1
40 IF X=0 THEN END ELSE GOTO 20
```

Unfortunately, there is something wrong with this program. It's possible that the value of X could be reduced to *less than* zero. When the expression in line 40 is evaluated it will be *false* if X= −1, or X= −1000! The program could continue forever or until you realize that it's stuck in a loop. This can be avoided by writing:

```
IF X<=0 THEN END ELSE GOTO 100
```

Now when the value of X is less than (<) or equal to (=) zero, the program ENDS.

The symbols for inequality can also be used with strings. For example, in the statement:

IF "A" < "B" THEN GOTO 100

The program jumps to line 100, because to BASIC the value of "A" is less than "B". In later chapters you will see how handy this can be when we want a program to sort a mailing list alphabetically.

LOGICAL OPERATORS IN IF . . . THEN STATEMENTS

There is another dimension to IF . . . THEN statements. Often a conditional statement contains more than one condition. We might say "If the customer orders 10 of item A *and* 2 of item B, then he receives a 10 percent discount." The *AND* used in this sentence is called a **logical operator.** In BASIC the logical operators used most often are **AND** and **OR.**

AND

The **AND** operator, when used in an IF . . . THEN statement, takes the form:

IF *(this expression is true)* **AND** *(this expression is true)* THEN *(this command is executed)* ELSE *(this command is executed)*

```
IF X= 1 AND Y=2 THEN GOTO 10 ELSE GOTO 500
```

In this statement the expression to the right of the THEN is executed *only if X=1* **AND** *Y=2*. Any other values of either variable cause the program to jump to line 500.

OR

The **OR** operator, when used in an IF . . . THEN statement, takes the form:

IF *(this expression is true)* **OR** *(this expression is true)* THEN *(this command is executed)* ELSE *(this command is executed)*

For example:

```
IF X=1 OR Y=2 THEN GOTO 10 ELSE GOTO 500
```

In this statement the program jumps to line 10 *if* X= **OR** Y=2. Only *one* of the expressions need be true for the program to jump to line 10. If neither expression is true the program jumps to line 500.

COMBINING LOGICAL OPERATORS

An IF . . . THEN statement can contain more than one logical operator. For example:

IF A=1 AND B=2 AND C=3 THEN GOTO 10 ELSE GOTO 100

This is a valid BASIC statement. In order for the program to jump to line 10 *all* three variables must be equal to the values in the expressions. When does a programmer use this? Any time three things need to be compared in one IF . . . THEN statement.

Logical operators can also be mixed in the same IF . . . THEN statement:

IF X=60 and Y=30 OR Z=15 THEN GOTO 200

In this case, if X=60 *and* Y= 30 then the program jumps to line 200. But if Z=15 the program jumps to line 200 regardless of the values of X and Y.

The number of expressions with logical operators in one BASIC line is limited only by the length of the line: 256 characters.

FOR . . . NEXT Loops

Remember that earlier in the chapter we introduced loops. In that section we showed how an IF . . . THEN statement limits the number of times that a program goes through a loop in a small program. This program is reproduced below:

```
10 X=1
20 PRINT X;
30 X=X+1
35 IF X = 10 THEN GOTO 50
40 GOTO 20
50 END
```

This program displays the numbers 1 through 10 only.

Well, another feature of BASIC, the **FOR . . . NEXT** loop, gives you a short-cut method of specifying the number of times that the program goes through a loop. This is one of those nice features of BASIC that makes programming easier for you.

The **FOR . . . NEXT** loop has two benefits. It allows you to control the number of times that the loop is executed and with very few lines of code it accomplishes a lot.

FOR . . . NEXT loops are used whenever a programmer wants a program to go through a loop a fixed number of times.

Here's how a **FOR . . . NEXT** loop works. A **FOR . . . NEXT** loop looks pretty complicated when you first see one. The **FOR . . . NEXT** loop takes the form:

FOR *(loop variable)* = *(start value)* TO *(end value)*
.
. lines of code to be
. executed within the loop
.

NEXT *(loop variable)*

When the program first enters the loop the *loop variable* is set equal to the *start value*. The succeeding lines of code are executed. Before the program encounters the **NEXT** instruction the value of the loop variable is compared to the *end value*. The start value and the end value are called the loop parameters. If the end value has been reached the program continues to the line following the **NEXT** instruction. But if the end value has not been reached control returns to the top of the loop, where 1 is added to the loop variable and the program goes through the loop again.

In the loop program lines 30–35, shown below, did this work.

```
30 X=X+1
35 IF X = 10 THEN GOTO 50
```

These functions are performed automatically by the **FOR . . . NEXT** loop.

The program in Figure 23 illustrates how a **FOR . . . NEXT** loop is used to print the same numbers 1–10 as the sample loop

program with the IF . . . THEN statement. When the program first enters line 30 the value of X is set equal to 1. Line 40 displays the value of X, 1, on the screen. Before line 50 the value of X, 1, is compared to 10, the ending loop parameter. Since the value of X is not yet equal to 10, control returns to line 30. The process is repeated until X=10. Figure 24 shows the output from this program.

```
10 'FOR...NEXT LOOP
20 CLS
30 FOR X = 1 TO 10
40 PRINT X;
50 NEXT X
60 END
```

Figure 23 FOR . . . NEXT Loop

```
 1  2  3  4  5  6  7  8  9  10
Ok
```

Figure 24 FOR . . . NEXT Loop Output

By the way, if you try to write a program with a **FOR** statement that doesn't have a matching **NEXT** statement BASIC will give you an ?**NF** error message. This indicates that you have a **FOR** without a **NEXT** error.

MORE ABOUT FOR . . . NEXT LOOPS

In Figure 23, 1 is added to X (the loop variable) each time the program goes through the loop. But what if you want to add 2 instead of 1? In the pre-FOR . . . NEXT loop program you would write:

```
30 X=X+2
```

To do this in a FOR . . . NEXT loop a programmer can tell the program to take two steps, so to speak. This is done with the **STEP** parameter. It's possible to define the amount that is added to the variable after each pass through the loop by specifying the **STEP** parameter at the end of the FOR line. An example is shown in the program below:

```
10 FOR X = 2 to 12 STEP 2
20 PRINT X
30 NEXT X
40 END

2 4 6 8 10 12
```

In the program above, the **STEP,** in line 10, is 2. This causes 2, 4, 6, 8, 10, and 12 to be output by the loop because X is incremented by 2 in each pass.

You don't have to *increase* the loop variable each time. If you want to count *down* the **STEP** can be a negative number, as shown below.

```
10  FOR X = 10 to 1 STEP -1
20  PRINT X
30  NEXT X
40  END

 10  9  8  7  6  5  4  3  2  1
```

This program starts out printing the number 10, and then subtracts 1 from X each time through the loop.

If a **STEP** is not specified BASIC assumes that the **STEP** is 1.

VARIABLE LOOP PARAMETERS

The starting and ending values for a FOR . . . NEXT loop, the loop parameters, can also be variables. Look at this variation of the FOR . . . NEXT loop:

```
10  A=3
```

```
20 B=33
30 C=3
40 FOR X = A TO B STEP C
50 PRINT X;
60 NEXT X
70 END

3 6 9 12 15 18 21 24 27 30 33
```

Lines 10–30 assign values to the variables A, B, and C. Then Line 40 uses these values as the parameters of the FOR . . . NEXT loop. This is an important concept because by using variables as loop parameters you can set the loop parameter *while the program is running*. How? Here's another example.

INPUT statements can be used in a program to let the user set the parameters of a FOR . . . NEXT loop, as shown in Figure 25. Lines 50, 60, and 70 use INPUT statements to get values for the loop parameters. Refer to the input screen, Figure 26, to see how the input prompts guide the user. Line 90 uses the values of A, B and C as the parameters of the FOR . . . NEXT loop. The loop produces the output shown in Figure 27.

```
10 ' FOR...NEXT LOOP WITH VARIABLE
PARAMETERS
20 CLS:PRINT
30 PRINT "THIS PROGRAM WILL PRINT THE
SERIES"
40 PRINT "OF NUMBERS THAT YOU SPECIFY."
50 INPUT "ENTER FIRST NUMBER...";A
60 INPUT "ENTER LAST NUMBER...";B
70 INPUT "ENTER INCREMENT BETWEEN NUMBERS
...";C
80 CLS:PRINT "HERE'S YOUR NUMBER SERIES
..."
90 FOR X = A TO B STEP C
100 PRINT X;
110 NEXT X
120 END
```

Figure 25 FOR . . . NEXT Loop Using Variables as Loop Parameters

```
THIS PROGRAM WILL PRINT THE SERIES
OF NUMBERS THAT YOU SPECIFY.
ENTER FIRST NUMBER...? 0
ENTER LAST NUMBER...? 500
ENTER INCREMENT BETWEEN NUMBERS...? 25
```

Figure 26 Input Screen

```
HERE'S YOUR NUMBER SERIES...
0  25   50    75   100   125   150   175   200
225   250   275   300   325   350   375   400
425   450   475   500
Ok
```

Figure 27 Output Screen

Almost all types of expressions, commands, and statements can be used in the program lines that are within a loop. You can use GOSUB to call a subroutine from within a FOR. . .NEXT loop but you must make sure the program RETURNS to within the loop. But sometimes programmers try to send a program through a loop a few times and then use GOTO to jump out before the FOR . . . NEXT loop is finished. This is the wrong way to use a FOR . . . NEXT loop. And that brings us to:

THE FOURTH RULE OF PROGRAMMING:

Always complete a FOR . . . NEXT loop. Never jump out of a FOR . . . NEXT loop with a GOTO statement.

BASIC is counting down each time the program goes through the loop. If you jump out of the FOR . . . NEXT loop before you're done you'll leave BASIC waiting for you to finish the loop.

NESTED LOOPS

If you've been reading straight through this section you are really doing great. And if you're referring to it for more information you must be doing great, too. So let's move on.

There's more to loops: FOR . . . NEXT loops can also be nested. That means that one FOR . . . NEXT loop can be placed within another. The program below uses two nested FOR . . . NEXT loops to display all the integers from 1 to 100. The inner loop is set apart by indenting lines 20–40.

```
10 FOR X = 0TO 90 STEP 10
20    FOR Y = 1 TO 10
30    PRINT X+Y;
40    NEXT Y
50 NEXT X
60 END
```

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19
20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67
68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83
84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
100
```

Enter this program in the Model 100 and try it. When line 10 is executed for the first time X is set to zero. Then in line 20 Y is set to 1. Line 30 calculates X+Y and sends the result, 1, to the screen. Line 40 compares the value of Y with the final value of the Y loop parameter, 10. Because Y is equal to 1 the program jumps back to the top of the Y loop, line 20. Y is incremented. The Y loop is executed again and again until the value of Y equals 10. At that point control falls through to line 50; the numbers 1 through 10 have been displayed. The value of X, 0, is compared to 90 and the program jumps back to the top of the X loop, line 10. Here X is incremented by the STEP of 10. This process continues until X=90. At that point the ending value of the X loop has been

reached, the integers 1 through 100 have been printed, and control falls through to line 60.

Please note, if your program printed the numbers one to a line you probably forgot to type the semicolon (;) at the end of line 30.

Here are three points to remember when using nested loops. They are not important enough to be made into rules:

- The outer loop controls the number of times that the inner loop(s) will be executed.
- The inner loop continues to execute until it reaches its own ending value. During that time the value of the parameters of the outer loop(s) remains unchanged.
- When the program exits a loop the value of the loop variable is incremented one STEP past the ending value. For example:

```
10 FOR X= 1 TO 10
20 PRINT X;
30 NEXT X
40 PRINT X

1  2  3  4  5  6  7  8  9  10
11
```

When the program *exits* the loop, in line 40 above, the value of X is 11, *not* 10.

READ and DATA Statements

The INPUT statement allows the user to enter data while the program is running. Data entered through INPUT statements usually changes each time the INPUT statement is encountered in the program.

There are times when a programmer wants a program to manipulate data that does not change often: product lists, price lists, inventory lists, customer lists, phone lists, and so on. In this case a programmer would enter such a list of data only once, *before* the program was run. The **READ** and **DATA** statements make this

possible. They are used together to assign values to variables. The **READ** statement gets these values from the **DATA** statements. **READ** and **DATA** statements follow the form:

READ *(variable)*

.
. program lines
.

DATA *(value), (value)*

Each time the program gets to the **READ** statement it reads data from the **DATA** statement. Each value in a **DATA** list must be separated by a comma. String values in **DATA** statements must also be enclosed in quotation marks. Let's look at some examples.

```
10 'PROGRAM USING READ AND DATA STATEMENTS
20 'TO COMPUTE THE AVERAGE OF TEN NUMBERS
30 FOR X=1 TO 10
40 READ N
50 TTL=TTL+N
60 NEXT X
70 AV=TTL/10
80 CLS
90 PRINT "The average is";AV
100 DATA 3,5,2,6,8
110 DATA 2,7,1,9,4
```

Figure 28 READ and DATA Statements

The program in Figure 28 uses the **READ** and **DATA** statements in a FOR . . . NEXT loop to compute the average of a list of 10 numbers.

The FOR . . . NEXT loop is set up to execute ten times, because there are ten numbers to be averaged. In line 40 the **READ** statement gets the value of N from the **DATA** statement in line 100. When the **READ** statement is *first* encountered the interpreter looks for the *first* piece of data in the *first* **DATA** statement. The **DATA** statements can be located anywhere in the program. More than one DATA statement can be used, as shown in lines 100–110.

On the *second* trip through the loop the **READ** statement is encountered again. The program has kept track of its place in the

DATA list and **READ**s the *second* value of N, adds it to the total, and continues until the loop is executed ten times. Remember, a microcomputer is doing all this and microcomputers are fast, so the entire process takes about one second. Line 70 calculates the average and line 90 displays the results, shown in Figure 29.

```
The average is 4.7
Ok
```

Figure 29 Output Screen

More than one variable, and more than one variable type, can be used in one **READ** statement provided that each variable is separated by a comma. Why would you want to use more than one variable type in a **READ** statement? Good question. How about if you wanted to **READ** a product name (using a string variable) and its product number (using a double precision variable.) The program in Figure 30 does just that: It uses more than one variable type in a **READ** statement to display customer information located in **DATA** statements.

```
10 FOR X = 1 TO 3
20 READ N,N$
30 PRINT "Product #";N;"-- ";N$
40 NEXT X
50 'DATA LIST
60 DATA 102,"Food Processor",
105,"Microwave Oven"
70 DATA 114, "Pasta Maker"
80 END
```

Figure 30 Program Using READ and DATA Statements

The **READ** statement in line 20 **READs** number N, (the product Number), and the string N$, (the product Name). The **DATA** statements list the data types in the same order as the variables in the **READ** statement. The output from this program is shown in Figure 31.

```
Ok
Run
Product # 102 -- Food Processor
Product # 105 -- Microwave Oven
Product # 114 -- Pasta Maker
Ok
```

Figure 31 Output Screen

READ . . . DATA statement combinations can be tricky. There are three things to watch when you use **READ** statements.

- You have to know the amount of data that you expect the program to **READ** and set up the loop accordingly. If the **READ** statement is executed after all the data items have been used the program will display an ?OD error message (Out of Data), which will make the program crash. You don't want your programs to crash.

  You know how much data you have because, as the programmer, you type in the **DATA** statements. If you are typing a customer list, for example, you just count the number of customers in your list. If you add or delete a customer, change the loop parameters.

- The sequence of data types in the **DATA** statements must follow the sequence of variable types in the **READ** statements. In the last program the **READ** statement in line 20 looked for a double precision variable, and then it looked for a string variable. (Aren't

you glad you read Chapter 2? Now you know all about double precision and string variables.)

```
20 READ N, N$
```

- When the **READ** statement "looks" at line 60 it will find the variables in the correct order. It's your job as programmer to organize the **DATA** statements.

```
60 DATA 102, "Food Processor"
```

- If the same **DATA** statements are to be **READ** more than once, use the **RESTORE** command to return the program to the first data item of the first **DATA** statement.

Program Flow and Algorithms

This second half of Chapter 3, as promised, will show you how to organize a problem or a task so that it can be solved by a BASIC program. We'll also create our first algorithm. But first things first.

The title of this chapter is Going With the Program Flow. Program flow is the order of execution of the lines of a BASIC program. It is the path that BASIC takes as it follows the twists and turns, the ins and outs, the straight sections and loops of your program. The BASIC statements covered in this chapter allow you to control program flow. Conditional and branching statements are the commands that can move a BASIC program away from its single-minded, one-step-at-a-time, straight-shooter ways and yet still keep program flow in control. And that is:

THE FIFTH RULE OF PROGRAMMING:

Control of program flow is the foundation for the algorithm. GOTO, GOSUB, IF . . . THEN, FOR . . . NEXT and READ . . . DATA statements are the building blocks of the algorithm.

We had some things to say about the algorithm earlier in this chapter. An algorithm is a finite series of steps that results in the solution to a problem. An algorithm can be simple or complicated.

And perhaps most important, we had the first and second rules of programming. Since we're about to create our first algorithm we repeat the first and second rules right here:

THE FIRST RULE OF PROGRAMMING:

When you want to write a BASIC program to solve a problem, first organize the problem so that it can be solved by a BASIC program.

Remember the three major sections of the first program in this book: input, calculation, and output? Well, in practical programming terms the first rule usually means organize the problem into three major sections: input, calculation, and output.

THE SECOND RULE OF PROGRAMMING:

If a task can't be understood and completed manually, it probably can't be programmed successfully.

This just means that you have to know what you're trying to do before you can do it.

Having read this far you are now ready to tackle your first algorithm. Since we have previously talked about a mailing list program, we're going to continue with it and create an algorithm to print mailing labels.

To create an algorithm to solve a problem in BASIC follow a two-step process:

- Describe the problem in words—make a list of the steps needed to solve the problem manually. (In other words, roll up your sleeves and apply the first and second rules.)
- Write lines of code to accomplish each step of the process. (In other words, roll up your sleeves even farther and apply everything else you have learned in this book.)

Fortunately, we're going to do this really slowly. We'll break the mailing label program down into "do-able" steps and then show how each step is put into action.

Writing the Mailing List Program

Step I: Describe the Problem.

These are the steps needed to successfully print mailing labels manually:

1. Make a list of the names and addresses.
2. Decide how many of the names and addresses you want to print.
3. Fetch the first mailing label.
4. Read the name, company name, street address, city/state, and zip code.
5. Print (type) the information from step 4 on the label.
6. Get a new mailing label.
7. Repeat steps 4–6 until you print the number of labels you want.

Step II: Write Lines of Code to Accomplish Each Step

The Mailing List Program, Figure 32, is the program that fulfills all the requirements of the algorithm. The lines that fulfill each step are clearly delineated by remark lines (note STEP 1, STEP 2, etc.) to help you compare the BASIC program to numbers 1 through 7 of the algorithm above.

```
10 'MAILING LIST PROGRAM
20 'STEP 1:
30 'MAKE A LIST OF NAMES AND ADDRESSES.
40       'DATA STATEMENTS ARE LOCATED AT
50       'BOTTOM OF PROGRAM TO ALLOW
60       'FOR ADDITION OF NEW NAMES
70       'WITHOUT RENUMBERING PROGRAM
LINES.
80 'STEP 2:
90 'DECIDE NUMBER OF NAMES & ADDRESSES TO
```

Figure 32 continues

```
PRINT
100 CLS
110 PRINT "      MAILING LIST PROGRAM"
120 PRINT:PRINT "HOW MANY NAMES AND
ADDRESSES ARE IN"
130 INPUT "THE LIST";E
140 'STEP 3:
150 'GET THE FIRST MAILING LABEL
160 CLS:BEEP
170 PRINT:PRINT "PUT MAILING LABELS IN
PRINTER"
180 PRINT
190 LINE INPUT "PRESS 'ENTER' TO
CONTINUE...";X$
200 'STEP 4:
210 'READ FIRST NAME, COMPANY NAME, STREET
ADDRESS,
220 'CITY/STATE AND ZIP CODE.
230 FOR X = 1 TO E
240 READ N$,CN$,ST$,CS$,ZP%
250 'STEP 5:
260 'PRINT THE INFORMATION ON LABEL
270 LPRINT N$
280 LPRINT CN$
290 LPRINT ST$
300 LPRINT CS$;" ";
310 LPRINT ZP
320 'STEP 6:
330 'GET A NEW LABEL
340 LPRINT
350 LPRINT
360 'STEP 7:
370 'REPEAT STEPS 4-6 UNTIL ALL LABELS ARE
PRINTED
380 NEXT X
390 CLS
400 PRINT "      MAILING LIST PROGRAM"
410 PRINT:PRINT E;"LABELS PRINTED"
```

Figure 32 continues

```
420 END
430 'DATA STATEMENTS:
440 DATA "Mr. S. Smith","Electro, Inc.",
"123 Main St.","Towners, NY",12345
450 DATA "Ms. J. Jones","Computer Corp.",
"111 Center St.","Valley, CA",12345
460 DATA "Mr. A. Name","Electronic
Supply","999 South Rd.","Northern,
NY",12345
470 DATA "Ms. N. Gineer","ABC Corp.","101
Second St.","Anytown, NY",12345
```

Figure 32 Mailing List Program

Remember, the algorithm is just the sequence of steps that the program follows. What we're going to look at now is the program that follows our algorithm.

The first step said to make a list of the names and addresses (see STEP 1.) The DATA statements in line 440–470 contain the list of names and addresses. This is a little tricky because even though making the list of names is number 1, the DATA statements that hold the names are placed at the bottom of the program.

By putting the list of names (DATA statements) at the *bottom* of the program you make it possible for new DATA statement lines to be added. If the DATA statements were at the top of the program you would have to start sneaking them in between other lines of the program, perhaps renumbering program lines to make room for the new DATA statements. What a drag that would be. The remarks in lines 30–70 make this clear.

One complete set of data for each name and address has been placed in each DATA statement. Note that proper syntax has been followed throughout: Each piece of string data is enclosed in quotation marks; each piece of data is separated by a comma; and the DATA statements list the data types in the same order as the variables in the READ statement.

Number 2 said to decide how many labels you wish to print. Lines 110–130 (in STEP 2) create a display that allows the user to enter the number of names and addresses that are in the data list. In line 130 the user sets the value of E, as shown in Figure 33, the ending parameter of the FOR . . . NEXT loop in line 230.

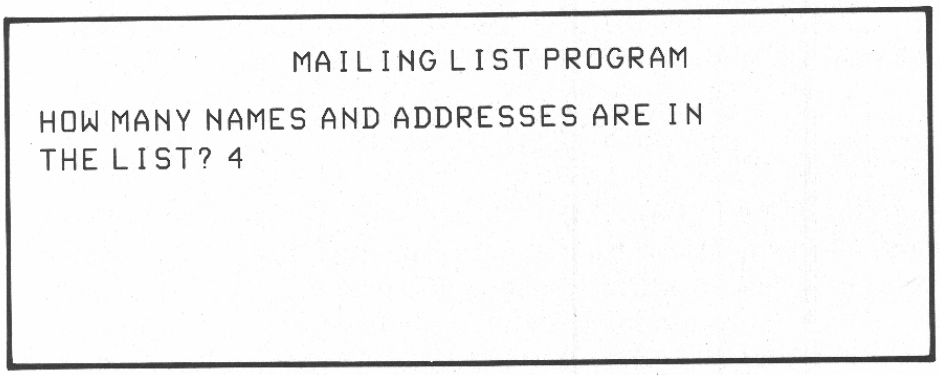


Figure 33 Input Screen

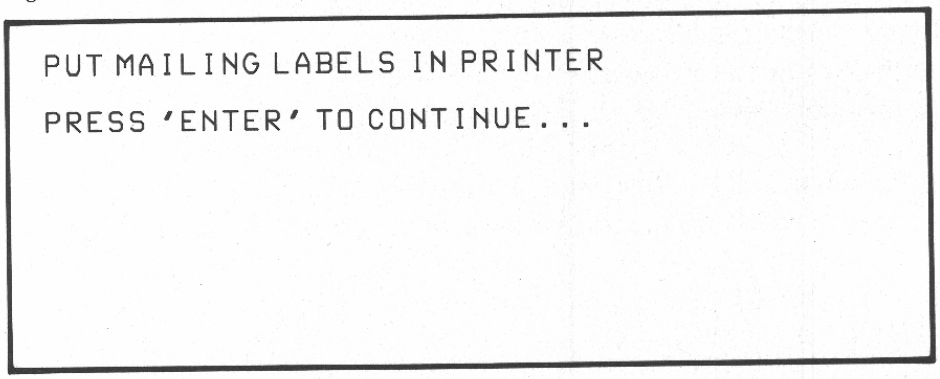


Figure 34 User-Friendly Feature

Number 3 stated that the first mailing label should be fetched. Lines 160–190 (STEP 3) sound the Model 100's built-in buzzer to remind you to put labels in the printer, as shown in Figure 34.

Number 4 stated that the first address should be read. Line 230 (STEP 4) establishes the FOR . . . NEXT loop parameters, using the variable E, set to 4 by the user in line 130.

Number 5 allows for the typing of the label. Lines 240–310 send the addresses to the printer. Line 240 (STEP 5) READs the five pieces of data that make up the complete label. The variables used are:

N$	Name
CN$	Company name
ST$	Street and number
CS$	City and state
ZP	Zip code

Lines 270–310 send this information to the printer using string variables and the LPRINT statement, with the form:

LPRINT *"text to be printed"*

The LPRINT statement works like the PRINT statement, but sends the information to a *printer* instead of the display screen Notice the spaces and the semicolon (;) at the end of line 300. This positions the zip code on the same line as the city and state.

Lines 340–350 (STEP 6) send two blank lines to the printer to move the next label into position. Since it is possible to get attached labels the computer even takes care of getting new labels. The number of line feeds to use with a given label must be determined by trial and error.

Line 380 (STEP 7), the NEXT statement, checks to see if all the labels have been printed. If not, the program jumps back to the top of the FOR . . . NEXT loop, and repeats the process.

```
Mr. S. Smith
Electro, Inc.
123 Main St.
Towners, NY 12345

Ms. J. Jones
Computer Corp.
111 Center St.
Valley, CA 12345

Mr. A. Name
Electronic Supply
999 South Rd.
Northern, NY 12345

Ms. N. Gineer
ABC Corp.
101 Second St.
Anytown, NY 12345
```

Figure 35 Mailing Labels from Mailing List Program

Figure 35 shows how printed labels would look using these data statements. Lines 400–420 END the program with an informative display. The user is told the number of labels, as shown in Figure 36, that have been printed. This is a frill but one most users appreciate.

```
                MAILING LIST PROGRAM

 4 LABELS PRINTED
Ok
```

Figure 36 User-Friendly Feature

Now that you've entered and run the mailing label program you really are making fantastic progress. Reading the last section of this chapter, which explains the **interrupt** command, is optional. You can return to it when interrupts are mentioned again. But if you want to read it now don't let me interrupt you.

Interrupt Commands

Interruptions. The word has a bad connotation. Nobody wants to be interrupted, or do they? What if someone reminds you of an important appointment. Or you receive an important telephone call from a friend? Is that an interruption? You may even want to be interrupted if you make a mistake that would affect future work.

Model 100 BASIC has special commands, called interrupt commands, that provide for these types of interruptions in BASIC programs. Interrupts are introduced here because they are conditional branching statements, as you'll see, and they affect program flow. This chapter will introduce the **ON TIME$** interrupt. The phrase **ON TIME$** interrupt, doesn't tell you very much and we'll explain it below.

Let's say you have many mailing labels to print and it will take all afternoon. But at 4:30 P.M. you want the Model 100 to

remind you of an important dinner appointment. The Mailing List Program, Figure 32, could be modified by adding the following program lines:

```
5 ON TIME$ = "16:30:00" GOSUB 480
6  TIME$ ON
10 'MAILING LIST PROGRAM
.
.     LINES 20-470 OF MAILING LIST
.     PROGRAM

480  'INTERRUPT SUBROUTINE
490 BEEP
500 CLS
510 PRINT "DON'T FORGET YOUR
APPOINTMENT . . ."
520 PRINT
530 LINE INPUT "PRESS 'ENTER' TO
RESUME PROGRAM . . .";X$
540 CLS:RETURN
```

Line 5 uses the **ON TIME$** command to tell BASIC to keep an eye on the clock. The **ON TIME$ interrupt** stops a BASIC program and calls a subroutine when the Model 100 clock reaches whatever time you specify. The **ON TIME$** command is only executed when that specified time is reached. Until that time is reached the program runs as if the interrupt command were not even there. **ON TIME$** follows the form:

ON TIME $ *(time you're looking out for)* GOSUB *(line number)*

where the *time you're looking out for* is in twenty four hour notation; hours:minutes:seconds.

We want the program to be interrupted at 4:30, so line 5 sets the ON TIME$ interrupt for 16:30:00 (4:30 pm):

```
5 ON TIME$ = "16:30:00" GOSUB 480
```

When the time reaches 16:30:00 (4:30 p.m.) the program jumps to the subroutine beginning at line 480.

Line 6 enables the interrupt. That is, it uses another command, **TIME$ ON,** to activate the clock-watching interrupt command. This is confusing since ON TIME$ and **TIME$ ON** appear to be the same command transposed, but they are not. **TIME$ ON** turns ON TIME$ *on*. Before you use an interrupt you have to enable it; turn it on. In this case you use the **TIME$ ON** command.

```
DON'T FORGET YOUR APPOINTMENT
PRESS 'ENTER' TO RESUME PROGRAM...
```

Figure 37 ON TIME$ Interrupt Screen

Lines 490–530 sound the buzzer and display a message, shown in Figure 37, to remind you of the 4:30 appointment. When you press ENTER a RETURN is executed.

The RETURN after an interrupt isn't the same as a RETURN used with a GOSUB. The interrupt RETURN is unique because it RETURNs the program to the line immediately following the last line executed before the interrupt subroutine was called. Thus, if label printing is interrupted printing resumes exactly where it was stopped by the interrupt. Not a single line of information is omitted.

Congratulations. You've reached another milestone on your road to fluency in the BASIC language. It might be a good idea now to try and write some short programs of your own. You've got a pretty good vocabulary of BASIC commands, and the Model 100 is at your command . . .

The next chapter, on arrays, adds to your ability to handle data with BASIC.

Chapter 4

BASIC Arrays: Handling Sets of Related Data

An array is BASIC's way of storing a list of data in memory. Computer programs often handle sets of related data such as sales data, customer data, employee data, and financial data. The ability of BASIC to compare, sort, store, or revise sets of related data makes it a very useful programming language. Using sets of related data the Model 100 computer can complete repetitive tasks such as sorting of mailing lists with accuracy and speed.

What does that have to do with arrays? The way we've been using variables to handle sets of data causes BASIC to recycle the variables. When BASIC needs a new piece of data it scraps the old value of a variable and substitutes the new value.

For example, the Mailing List Program, Figure 32 in Chapter 3, prints four sets of name-and-address data. In the READ statement in line 240:

```
240 READ N$,CN$,ST$,CS$,ZP
```

variables N$, CN$, ST$, CS$ and ZP are recycled. Each time new data is read from the DATA statements the variables pick up the new data and discard the old. Out with the old, in with the new.

But there are times when the old *and* the new are both needed.

There are disadvantages to using individual variables in this way to represent sets of related data. The main disadvantage is that if you want to represent a large set of data you would have to think up a lot of variable names, keep them all straight, and then figure out a way to use them.

BASIC provides another better way of manipulating sets of related data—the arrays.

When you use an array in a BASIC program, BASIC stores all the data as a list in memory. Then the program can use any or all of the data when it is needed but it doesn't need to discard any values because there is a place for each value in the list. When you use an array, it takes the form:

variable name *(subscript number)*

The subscript number in the array serves the same purpose as the product number in the example below. It indicates the item in the list that is being used at any one time.

Let's say that you have a price list that contains five products. You could make up individual variable names to store each product price: P1 = product 1, P2 = product 2, . . . and so on.

Here, for instance, is what a list that stores the prices of five products might look like:

Product Number	Price
P1	200
P2	500
P3	600
P4	100
P5	550

But pretty soon you could run out of either variable names or patience keeping track of everything. Well, when you use an array to store your price list it *automatically* creates a unique variable for each price and keeps track of it. Not bad. Not only that, it gives you a fast, simple method of getting to each and every price without juggling ridiculous variable names like P1, P2, and so on.

Here, for instance, is how an array with five items of data would store the price list used in the previous example. Each element of the array stores one piece of data in the list:

Product Number	Price
P(0)	200
P(1)	500
P(2)	600
P(3)	100
P(4)	550

Individual variables in an array are sometimes called elements. Each element holds one piece of data. The array contains these five elements:

P(0), P(1), P(2), P(3), P(4)

Each element of the array (P(0), P(1), P(2), etc.) is a variable and can hold one piece of data. You refer to individual array variables by specifying their subscript number.

Some more details about arrays: in the element P(1), 1 is the subscript, *not* the value. In P(2), 2 is the subscript. P(1) and P(2) are unique variables and can hold two separate pieces of data.

Each element of an array has all the other characteristics of non-array variables. It can store a number or a string. It can have one of the four data types (double precision, single precision, integer or string). The variable name used for the array variable must conform to all the rules for non-array variables. All of the following are valid array variable names:

ZP!(10) NM$(10) TL(10) A(10)

One more thing, in an array the subscript itself can be a variable. These are all valid arrays:

ZP!(X) NM$(B) TL(N) A(X)

In the array A(X), X is the subscript. In the expression:

A(X)=2*X

the element of the array that is used will depend on the value of X. The following program uses the array A(X) to produce a series of numbers, store them in memory, and display them.

```
10 FOR X = 1 TO 5
20 A(X) = 2*X
```

```
30 NEXT X
40    FOR X = 1 TO 5
50    PRINT A(X)
60    NEXT X
70 END

2 4 6 8 10
```

In the first loop, line 20 uses the array A(X) in an assignment statement. The first time through the loop X=1, and the array element used is A(1). Because line 20 is an assignment statement the value of A(1) is set to 2*X, or 2. After five trips through this loop five elements of the array are assigned five new values:

A(1)=2 A(2)=4 A(3)=6 A(4)=8 A(5)=10

These values are stored in memory.

The parameters of the second FOR...NEXT loop are also 1 to 5. Line 50 uses the array once again to display the values of A(1) through A(5)—2,4,6,8, and 10.

Note that in line 50 the BASIC program takes the values of the array *directly from the Model 100's memory*. The first FOR...NEXT loop (lines 10–30) stores the values in memory and the second FOR...NEXT loop (lines 40–60) uses them.

This is an essential feature of arrays. Once the values have been assigned they are stored in memory until they are needed by the program. They can be retrieved sequentially, using FOR...NEXT loops, or they can be accessed individually. We'll see how to use the values stored in arrays in the sample programs, but first we have to see how to dimension an array.

Dimensioning Arrays

You can tell BASIC how large to make the array. This is called dimensioning the array. To dimension an array is to tell BASIC how many items will be in the list of data stored by the array. A **DIM** statement takes the form:

DIM array variable name *(number of elements in array-1)* where the number of elements in the array is the number of pieces of data that the array can store.

We can create an array, P(5), that contains six elements (and can hold six pieces of data). We need to use the **DIM** statement:

DIM P(5)

In **DIM** statements the value between the parentheses, **DIM** P(5), is *not* a subscript. It tells BASIC how many elements you want the array to contain. The array that **DIM** P(5) creates will have these elements:

P(0) P(1) P(2) P(3) P(4) P(5)

If you don't specify the size of an array BASIC, by default, will give it the dimension of 10.

Remember, the array dimensioned with this dimension statement:

DIM A(10)

will have *11* elements because the first subscript of the array is always zero. The elements of an array dimensioned A(10) will be:

A(0) A(1) A(2) A(3) A(4) A(5)
A(6) A(7) A(8) A(9) A(10)

To set up an array for the variable NM$ with 26 elements use the dimension statement:

DIM NM$(25)

More than one array can be dimensioned in one DIM statement if each array is separated by a comma:

DIM A(12), B(25), C(100)

In a **DIM** statement the size of the array can be a variable. For example in the statement:

DIM NM$ (E)

the size of the array will depend on the value of the variable E.

If you try to dimension an array that BASIC thinks is too big

it will give you an **?OV** error message. **?OV** means **OV**erflow, and overflow means that BASIC can't handle the number of items you are trying to use.

Sorting Arrays

One reason to use arrays is sorting. Lists of data stored in arrays are easy to sort either numerically or alphabetically. Using arrays, for instance, makes sorting a mailing list alphabetically and by zip code quite easy. You can then print out a set of mailing lists in zip code order, and one in alphabetical order. The zip code grouping makes the post office happy and the alphabetical listing makes it easy to find customer names.

```
10 'MAILING LIST PROGRAM USING ARRAYS
20 'TO SORT DATA ALPHABETICALLY AND BY ZIP
CODE
30 CLS:PRINT "    MAILING LIST PROGRAM":
PRINT
40 PRINT "HOW MANY NAMES AND ADDRESSES ARE
IN"
50 INPUT "THE LIST";E
60 '---------------DIMENSION ARRAYS
70 DIM NM$(E),CN$(E),ST$(E),CS$(E),
ZP(E),PC%(E)
80 '----------------READ DATA
90 FOR X= 1 TO E
100 READ NM$(X),CN$(X),ST$(X),CS$(X),
ZP(X), PC%(X)
110 NEXT X
120 '----------------SORT DATA
ALPHABETICALLY BY CO. NAME
130 FOR X=1 TO E-1
140     FOR Y=X+1 TO E
150     IF CN$(X)>CN$(Y) THEN GOSUB 320
160     NEXT Y
170 NEXT X
180 W$="ALPHABETICALLY BY CO. NAME"
```

Figure 38 continues

```
190 GOSUB 430
200 '----------------SORT DATA
NUMERICALLY BY ZIP CODE
210 FOR X=1 TO E-1
220     FOR Y=X+1 TO E
230     IF ZP(X)>ZP(Y) THEN GOSUB 320
240     NEXT Y
250 NEXT X
260 W$="NUMERICALLY BY ZIP CODE"
270 GOSUB 430
280 CLS
290 PRINT
300 PRINT "PRESS KEY F4 TO RERUN PROGRAM"
310 END
320 '--------------SORTING SUBROUTINE
330 '  STORE HIGH VALUE
340 NM$=NM$(X):CN$=CN$(X):ST$=ST$(X)
350 CS$=CS$(X):ZP=ZP(X):PC%=PC%(X)
360 '  SWITCH HIGH AND LOW VALUE
370 NM$(X)=NM$(Y):CN$(X)=CN$(Y):ST$(X)
=ST$(Y)
380 CS$(X)=CS$(Y):ZP(X)=ZP(Y):PC%(X)=
PC%(Y)
390'  SWAP LOW VALUE FOR STORED VALUE
400 NM$(Y)=NM$:CN$(Y)=CN$:ST$(Y)=ST$
410 CS$(Y)=CS$:ZP(Y)=ZP:PC%(Y)=PC%
420 RETURN
430 '--------------OUTPUT SUBROUTINE
440 CLS:PRINT "    MAILING LIST
PROGRAM":PRINT
450 PRINT "DO YOU WANT A LISTING PRINTED"
460 PRINT W$;"?"
470 LINE INPUT "(Y/N)...";X$
480 IF X$="Y" OR X$="y" THEN GOTO 490 ELSE 600
490 BEEP
500 PRINT "PLEASE WAIT...LISTING
PRINTING"
```

Figure 38 continues

```
510 LPRINT "LISTING ";W$
520 LPRINT "CO. NAME";TAB(21);"ADDRESS";
TAB(50);"ZIP CODE;TAB(60);"NAME";
TAB(75);"P.CD."
530 LPRINT "-----------------------------
-----------------------------------------
------------"
540 FOR X=1 TO E
550 LPRINT CN$(X);",";TAB(21);ST$(X);
CS$(X) TAB(50);ZP(X);TAB(60);
560 LPRINT NM$(X);TAB(75);PC%(X)
570 LPRINT
580 NEXT X
590 CLS:PRINT "    MAILING LIST PROGRAM
":PRINT
600 PRINT "DO YOU WANT LABELS PRINTED"
610 PRINT W$;"?"
620 LINE INPUT "(Y/N)...";X$
630 IF X$="Y" OR X$="y" THEN GOTO
    640 ELSE 730
640 BEEP
650 PRINT "PLEASE WAIT...LABELS PRINTING"
660 FOR X=1 TO E
670 LPRINT NM$(X);TAB(25)
680 LPRINT CN$(X)
690 LPRINT ST$(X)
700 LPRINT CS$(X);"  ";ZP(X)
710 LPRINT:LPRINT
720 NEXT X
730 RETURN
740 '--------------DATA STATEMENTS:
750 DATA "Mr. S. Smith","Electro,
Inc.","123 Main St.","Towners,
NY",12345,3
760 DATA "Ms. J. Jones",Computer Corp.",
"111 Center St.","Valley, CA",10001,4
770 DATA "Mr. A. Name","Electronic
Supply","999 South Rd.","Northern,
```

Figure 38 continues

```
NY",30005,1
780 DATA "Ms. N. Gineer","ABC Corp.","101
Second St.","Anytown, NY",90045,2
```

Figure 38 Mailing List Program Using Arrays to Sort Mailing List

The Revised Mailing List Program, Figure 38, uses arrays to sort the name and address data by company name and by zip code. It gives the user the option to print a listing of the data or to print mailing labels of the sorted customer information. The sorting routines in lines 130–170 and 210–250 would not be possible without arrays.

```
          MAILING LIST PROGRAM

HOW MANY NAMES AND ADDRESSES ARE IN
THE LIST? 4
```

Figure 39 Input Screen

Fasten your seat belts. This program uses nearly all the commands that we've learned up to this point. We explain all the new commands below, but note that we no longer explain commands you are now familiar with. Let's see how Figure 39 puts arrays to work.

Lines 30–50 create the display in Figure 39 to ask the user the number of DATA statements to be READ. The value entered is assigned to the variable E. The variable name E was used because it stands for the number of *elements* in the arrays. This variable will be used more than once by the program.

The dimension statement in line 70 uses the value of E to dimension the arrays that hold the data. These arrays are used both for input and output. They are:

NM$(E)	Name
CN$(E)	Company name
ST$(E)	Street and number
CS$(E)	City and state
ZP(E)	Zip code
PC%(E)	Product code

All six arrays, and three different variable types, are dimensioned in one DIM statement in line 70:

```
70 DIM NM$(E),CN$(E),ST$(E),
CS$(E), ZP(E),PC%(E)
```

There are four arrays for string information:

NM$(E) CN$(E) ST$(E) CS$(E)

one for double precision numbers:

ZP(E)

and one for single precision numbers:

PC%(E)

The size of the arrays depends on the value of E from line 50, because E is used to dimension the arrays.

Lines 90–110 puts the data FROM the DATA statements into the arrays. Line 90 uses the value of E in a FOR . . . NEXT loop because you want the program to go through the loop as many times as there are names and addresses in the DATA statements. Line 90 READs the information in the DATA statements and stores this data in the arrays dimensioned above.

Sorting Routine

If you want to alphabetize a list of names manually you compare the name at the top of the list with all the other names in the list.

If the top name has a higher alphabetic value, you switch its position with the name farther down the list. The letter Z for instance, has a higher value than the letter A. This is repeated until all the names are sorted alphabetically. This is also an algorithm that a BASIC program can use to sort a list of names.

This algorithm is used in lines 130–170 to compare the value of the company name at the top of the list with every other company name in the list. The program swaps values if necessary to move the lowest values toward the top of the list, just the way you'd do it by hand.

BASIC does this by actually comparing their position in the table of ASCII codes (appendix A). For example, the letter B is ASCII number 66. The letter A is ASCII number 65. In the statement:

```
IF "A"<"B" THEN GOTO 100
```

the program jumps to line 100 because 65 is less than 66.

But "A" and "B" are only one letter each. That's easy. What if BASIC has to compare *words*? When words or phrases are compared BASIC compares them one letter at a time. They are either equal or one has a higher value than the other. When the words ABASH and ABATE are compared BASIC breaks them down like this:

```
A   B   A   S   H     A   B   A   T   E
65  66  65  83  72    65  66  65  84  69
```

BASIC compares these strings one character at a time. The first three characters are the same (ABA) and their ASCII values are equal (65, 66 and 65). But the fourth characters of these words are not the same and their ASCII values are not equal. S has an ASCII value of 83, while T has an ASCII value of 84. Therefore in a sorting routine BASIC can discern that ABATE should occupy a position farther down an alphabetized list than ABASH. Of course, you have to supply the sorting algorithm for BASIC to follow.

Lines 130–170 contain just such a complicated sorting routine. You can use this routine two ways: the easy way and the hard way.

The easy way: If you ever have to write a program that sorts something you can pretty much copy what we have here.

The hard way: You follow the discussion of lines 130–170 and understand how it works. For now, if I were you I'd go the easy way.

Lines 130–170 use two nested loops to sort the sets of data alphabetically by company name. The loop parameters are set so that the Y FOR . . . NEXT loop is always "one step ahead" of the X FOR . . . NEXT loop.

```
130 FOR X=1 TO E-1
140 FOR Y=X+1 TO E
```

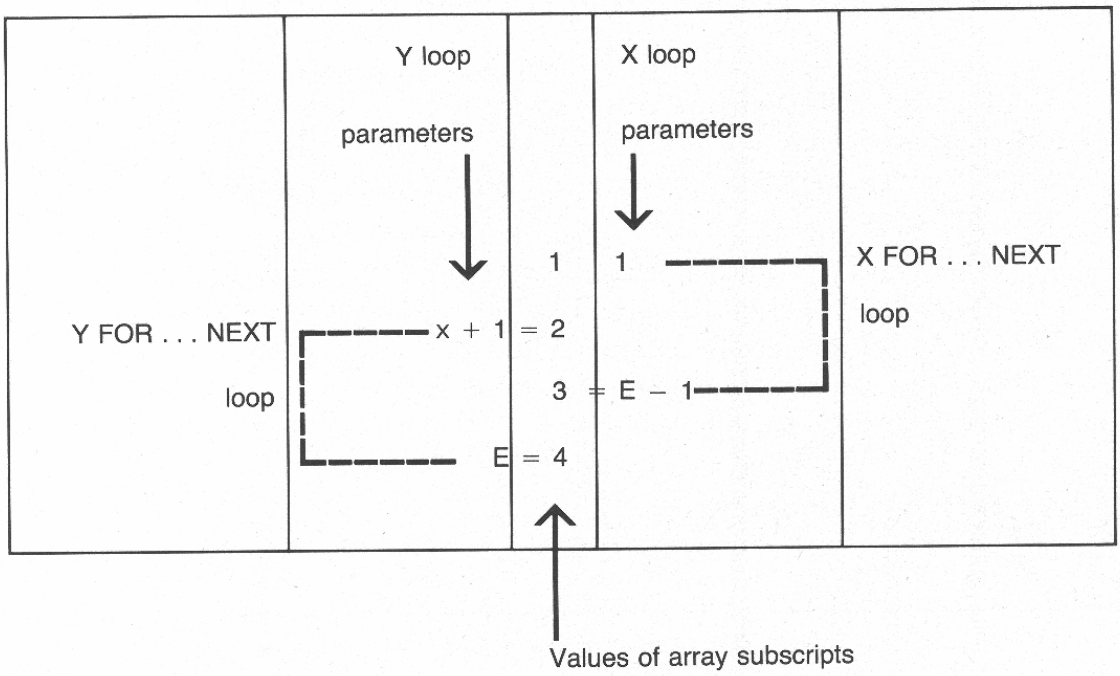


Figure 40 Two Nested Loops

The diagram in Figure 40 shows how these two nested loops are related. In line 150 CN$(X) is compared to CN$(Y):

```
150 IF CN$(X)>CN$(Y) THEN GOSUB 320
```

IF the value of CN$(X) is greater than CN$(Y), THEN the subroutine beginning at line 320 is called. If the X element already

has a lower value than the Y element, the program proceeds to line 160 and the subroutine is not called.

The sorting subroutine, lines 320–420, switches the values of the X and Y array elements with a three-step process:

1. First, in line 340–350, all six of the higher values are stored in *temporary variables*. For example, in line 340, the variable NM$ is *not* the same variable as NM$(X). It's a temporary variable used to store the higher value while the positions are being switched in the array.
2. Then in lines 370–380, the values of the X and Y elements are switched.
3. And last, in lines 400–410 the low value is swapped for the value stored in the temporary non-array. The RETURN sends the program back to line 160. The FOR . . . NEXT loop continues until all the values have been compared and the data is sorted from A to Z.

When the program has finished the nest of FOR . . . NEXT loops in lines 130–170, line 180 assigns the string ALPHABETICALLY BY CO. NAME to the string variable, W$. This variable will be used in the output routine. Then line 190 calls the output subroutine and the program goes down to line 430.

Output Subroutine

```
                MAILING LIST PROGRAM

DO YOU WANT A LISTING PRINTED ALPHABETICALLY
BY CO. NAME?
(Y/N)...Y
```

Figure 41 Input Screen

```
LISTING ALPHABETICALLY BY CO. NAME
CO. NAME              ADDRESS                          ZIP CODE  NAME           P.CD.
--------------------------------------------------- ------------------------------
ABC Corp.,            101 Second St.Anytown, NY        90045     Ms. N. Gineer  2
Computer Corp.,       111 Center St.Valley, CA         10001     Ms. J. Jones   4
Electro, Inc.,        123 Main St.Towners, NY          12345     Mr. S. Smith   3
Electronic Supply,    999 South Rd.Northern, NY        30005     Mr. A. Name    1

LISTING NUMERICALLY BY ZIP CODE
CO. NAME              ADDRESS                          ZIP CODE  NAME           P.CD.
--------------------------------------------------- ------------------------------
Computer Corp.,       111 Center St.Valley, CA         10001     Ms. J. Jones   4
Electro, Inc.,        123 Main St.Towners, NY          12345     Mr. S. Smith   3
Electronic Supply,    999 South Rd.Northern, NY        30005     Mr. A. Name    1
ABC Corp.,            101 Second St.Anytown, NY        90045     Ms. N. Gineer  2
```

Figure 42 Listing of Sorted Mailing List

The output subroutine, lines 430–730, prints a listing of the data, mailing labels or both. Figure 41 shows the screen created by lines 440–470. The user's response determines whether the alphabetically sorted data is sent to the printer. Figure 42 shows the two listings produced by the program. The data is listed alphabetically by company name and then numerically by zip code.

```
Ms. J. Jones
Computer Corp.
111 Center St.
Valley, CA      10001

Mr. S. Smith
Electro, Inc.
123 Main St.
Towners, NY     12345

Mr. A. Name
Electronic Supply
999 South Rd.
Northern, NY    30005

Ms. N. Gineer
ABC Corp.
101 Second St.
Anytown, NY     90045
```

Figure 43 Mailing Labels, Sorted by Zip Code

The mailing labels in Figure 43 are printed in zip code order. Why sort data in the first place? Sorting can help you understand and use the data. A mailing list in alphabetical order can be a lot easier to use than one in random order. And if the BASIC program does the sorting, so much the better.

Other Program Features

The string variable W$ carries its share of the load in this program. It's used in line 460, in a screen display; in line 510, in a line sent to the printer; and in line 610, in another display. Since the value of this variable is assigned as the program exits the sorting routines, the display, or listing created by W$ is changed each time.

The Mailing List program contains user-friendly features including the displays in the output subroutine, and the message PLEASE WAIT . . . LISTING (or LABELS) PRINTING, in lines 500 and 650. The program even ENDs with a message instructing the user to PRESS F4 TO RERUN PROGRAM, as shown in Figure 44, if the user wants to print a second set of labels.

```
PRESS KEY F4 TO RERUN PROGRAM
Ok
```

Figure 44 User-Friendly Feature

The second sorting routine (lines 210–250) uses the same algorithm but substitutes the zip code variable, ZP(X), for the name variable, NM$.

To test your understanding of the algorithm, rewrite the routine in lines 210–270 to sort the data numerically by *product code*. Here are some hints on how to do it:

- The array for the product code is PC%(X) (the variables for this program are listed on page 113). Follow the algorithm used in lines 210–250 substituting PC%(X) for ZP(X).
- Use PC%(X) and PC% (Y) in the IF . . . THEN statement in line 230.

- And assign the string NUMERICALLY BY PRODUCT CODE to the variable W$ in line 260.

Then run your program and see how it works.

There are a few key points to remember when using arrays:

- An array can only be dimensioned once in a program. If the DIM statement is executed more than once, the program crashes and BASIC displays a **?DD** error message, meaning there is a Double Dimensioned array.
- Dimensioning an array reserves a place in memory to store each element of the array. Once values are stored in the array they are immediately available to the program.
- When an array is dimensioned a finite number of spaces are saved in memory for the array elements. If the program tries to use an array element that does not exist, the program will crash. For example, if an array is dimensioned A(10) and the program tries to execute this instruction:

  A(12) = 26

  the program will crash because A(12) does not exist and BASIC has no place to store the value.
- The number of elements in an array is one more than the number used in the dimension statement because the first element has the subscript of zero. DIM A(10) produces an array with 11 elements; A(0)-(10).
- When you want to sort a set of related data, store it in an array and use the algorithm in Figure 39.

With the addition of arrays the Mailing List program attains a new level of efficiency. The program can process information stored in arrays, as in the sorting routine, and reuse it.

But we still have a problem. When we turn off the power of the Model 100 the data values produced by the program and stored in the array are wiped out, unless we have created files. Files are the subject of the next chapter.

Chapter 5

Sequential Files and the Model 100's Secret Filing Cabinet

This chapter introduces sequential files. As you'll learn in this chapter, BASIC programs use files to store information in the Model 100 in much the same way that a file cabinet is used to store information in an office. This concept is explained in this chapter.

After you understand how to use files, you will have a new way to feed information to your programs, and to store and use information produced by your programs. You'll learn how BASIC programs create sequential files to store data, and you can create them with the TEXT word processor. This may sound confusing right now but it will all be explained as the chapter progresses.

We said that programs manipulate information. So far in this book you've learned two ways to feed a program information: typing at the keyboard in response to INPUT or LINE INPUT statements; and DATA statements. There are disadvantages to both of these methods of feeding data to the program.

First, when you enter information at the keyboard it vanishes when the program is over. So it has to be reentered each time the program is run. There's a limit to the volume of information you can ask anyone to type at the keyboard. The IRS's computer pro-

gram has information on *millions* of taxpayers. You know that the IRS does not retype all these names every morning. There must be a better way.

Second, DATA statements can hold information for the program to use. This solves the problem of having to type in the data every time you use the program. But it still leaves one problem unsolved: Only the programmer can add to the list of DATA statements. Practical computer programs need a way to add to their list of data *while the program is running*. For example, when you order an airline ticket, the airline's computer adds your reservation to the program's data immediately. This can't be done with DATA statements.

BASIC's answer to these problems is the file. Specifically, Model 100 BASIC uses sequential files. What's that, you ask? This entire chapter answers your questions, and it starts by answering the question:

What Is a File?

Files are used in business every day. You use file folders to store typed or handwritten information. Model 100 BASIC uses files to store sets of information electronically.

Files store information and BASIC excels at handling information. With BASIC you can create files for any type of information that can be expressed in numbers and letters, including customer names and addresses, technical data, order information, telephone numbers, price lists, parts inventories, and more.

Both manual and computerized files function in about the same manner. But with the Model 100 you store and retrieve information with BASIC commands. This chapter describes files, explains their use, and shows you how to put your information in them and get it out again.

Input/Output

Storage and retrieval of files is one type of input/output. Input/Output, often abbreviated I/O, is a technical term for the process of communicating with the computer. Input means sending infor-

mation to the computer. Output means receiving information from the computer.

For example, we send information to the Model 100 whenever we type at the keyboard. We receive information from the Model 100 whenever it is displayed on the screen or sent to the printer.

I/O also includes loading and saving BASIC programs. Loading a program is Input and saving a program is Output.

Devices

The Model 100 is so compact that it obscures the original idea of input/output. In the old days, when the computer was in one room and the keyboard was in the next room, it made sense to talk about input and output. You went into the room with the keyboard to do input. You went into a third room to get your output from the printer.

The Model 100 is so small that it appears to be one discrete operating unit. Who wants to talk about input/output when you can put the whole computer—including keyboard, display, memory—into your briefcase?

But the Model 100 is a computer with an attached keyboard, display, memory, and modem. There's a great jargon word for the keyboard, display, and so on: I/O device. A device is a specific functional unit outside of the computer. Devices include the display screen, the printer, the sound generator, and the keyboard. The Model 100's RAM memory, cassette interface, modem interface, and RS232C communications interface are also devices. (An interface is jargon for the electronic and electro–mechanical parts that allow you to connect two electronic units.)

The illustration in Figure 45 shows the types of devices available on the Model 100. BASIC gives you a way to send information to the computer and to get information from the computer. A file is just another way to get information in and out of the Model 100. Let's see what that means.

Input/output is organized into two main categories: *immediate* I/O and *file* I/O. Immediate I/O includes the information sent to the display screen, the printer, and the sound generator or that received from the keyboard.

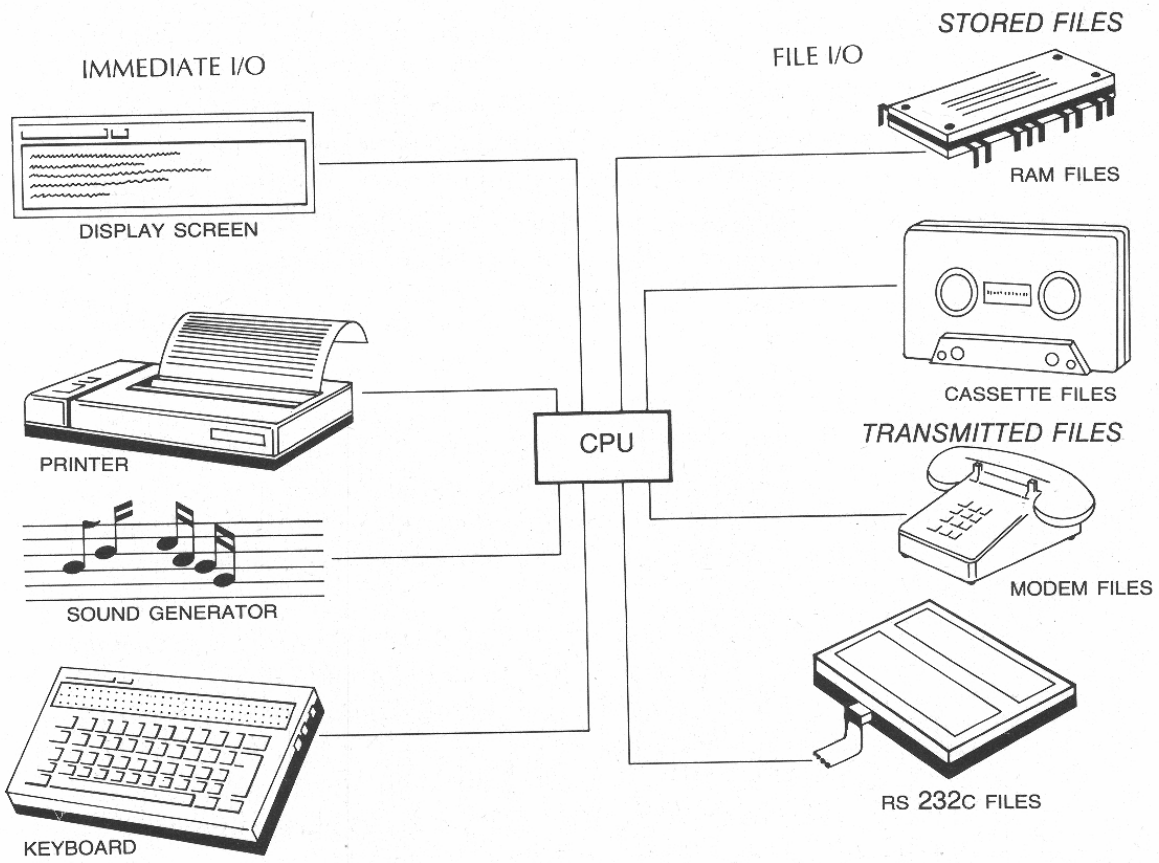


Figure 45 Types of Input/Output on the Model 100

BASIC handles immediate I/O for you. When you write a statement such as:

LPRINT "NAME"

BASIC sends the string, NAME, to the printer, and the printer produces the output you need. The same can be said of the command:

PRINT "NAME"

File I/O is more demanding. File I/O requires more attention and control by the programmer. We'll see why as files are covered in detail in this chapter.

There are two types of files: program files and data files. Program files include the programs supplied with the Model 100 (BASIC, TEXT, TELCOM, ADDRSS, and SCHEDL) and the BASIC

programs that you create. Data files include text files prepared with the TEXT word processor and data stored by BASIC programs. We'll cover them one at a time, program files first.

Program Files

When you look at the Model 100's main menu you see a display that is similar to Figure 46. The main menu displays the names of the files that are stored in the Model 100. The first five programs, listed without extensions, are the built-in programs. They're ready to run and can't be changed.

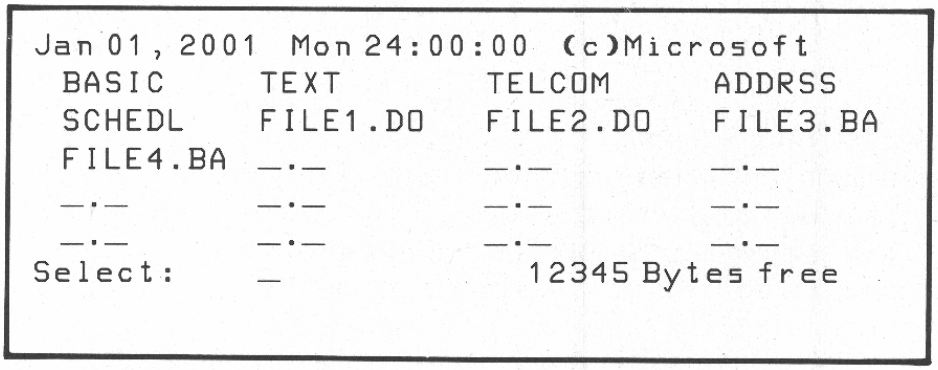


Figure 46 Model 100 Main Menu Showing Files

The files displayed with the extension .BA are files that are created when programs are saved with the SAVE command. They are called BASIC programs files. SAVE was included in Chapter 0 but—as is true of everything else you can learn about computers—there's more to it. The next few sections of this chapter cover in detail the many commands you can use to save and load program files. The second half of the chapter shows how to create data files.

Saving Program Files

BASIC program files contain the program lines (line numbers and commands) of BASIC programs. BASIC program files are created when you save a BASIC program. Programs can be saved in the Model 100's RAM memory, or externally on cassette tape.

When you SAVE a file in RAM (Random Access Memory) it is stored electronically. The smallest amount of data that can be stored by a BASIC program is one byte. One byte is not very much storage; one ASCII character occupies one byte of RAM. One kilobyte is equal to 1024 bytes. The designation 8K means that there are 8 kilobytes of RAM storage. 32K means 32 kilobytes.

"This is a string of text."

would occupy 26 bytes, because there are 25 characters, including spaces, plus one byte for the carriage return at the end of the sentence. Surprised that the carriage return takes up a byte? You can think of bytes like this: Every time you press a key to create a BASIC program you are storing one byte. Similarly, every time you press a key to create a document with the TEXT word processor you are storing one byte.

The Mailing List program, Figure 32, occupies about 1340 bytes of RAM, or about 1.3K. A short memo with 150 words contains about one kilobyte. (That's why we said in the Introduction that any *one* program in this book can run on the 8K Model 100 as long as there aren't a lot of other files stored in RAM. A couple of program files and a few memos can use up all your RAM memory. And BASIC's like an elephant, good for nothing without memory.) Why are we talking about memory? When you save a BASIC program, it is stored in memory along with anything else that you may have stored previously.

To save the current BASIC program (the current program is the program that has just RUN or been created while in BASIC command mode) use the SAVE command:

SAVE "RAM:*FLNAME*.BA"

FLNAME is the six-character filename of your choice. Filenames created by you can be up to six characters in length, must begin with an alphabetic character, and have an extension. The extension is the two-letter suffix following the period. RAM: tells BASIC to store the file in RAM and the extension BA indicates that it is a BASIC program file. The command:

SAVE "FLNAME"

will produce the same result. Unless you specify otherwise BASIC

assumes that the destination is RAM and adds the proper extension for you. Saving a program file using the name of a file that already exists erases the existing file.

When you create or edit a BASIC program it is composed of ASCII character codes. But when you save a BASIC program it is converted from ASCII characters to a compressed digital format. This is advantageous for most purposes because it saves space. But it is sometimes necessary to save a BASIC program in ASCII format (an example is given in the programming debugging chapter). To do so add a comma and the letter A (,A) to the end of the SAVE command. Programs can be SAVEd in ASCII format using the SAVE command:

SAVE "RAM:FLNAME.DO",A

When a program file is saved with the ,A option the filename must have the .DO extension.

Loading Program Files

There are two ways to load a BASIC program.

1. From the MAIN MENU, place the cursor over the name of the program file that you want to run and press ENTER. The program will load and run.

2. From within BASIC type the command:

LOAD "RAM:FLNAME.BA"

The program will LOAD, but it will not RUN until you enter the RUN command. As with the SAVE command, LOAD "FLNAME" may be substituted for the long version of the command.

FILES, NAME AND KILL

From within BASIC you can display a listing of files in RAM by pressing function key F1, or by typing the command:

FILES

To change the name of a file use the command:

NAME "FLNAME.EX" AS "NUNAME.EX"

FLNAME.EX is the current filename and extension. NUNAME.EX will be the new filename and extension. The quotation marks are required.

You can delete a file using the command:

KILL "FLNAME.EX"

When you kill a program file you regain the memory space that it occupied. But a **KILL**ed file is like yesterday. It's *gone forever and can't be retrieved.* Before you kill a file make sure you either don't need it anymore or have a backup on cassette tape.

Using Cassette Tape to Save Files

The main advantage of cassette tape is to store backup copies. And program files saved on cassette tape can be loaded into other Model 100s, allowing you to share your programs. (If you have the Disk/Video Interface, we'll cover diskette files in Chapter 9. But this chapter is a prerequisite to Chapter 9, so stay with us.)

There are disadvantages to cassette tape, as you'll see if you work with it. Most cassette tape players are almost as big and as heavy as the Model 100. Tape players are slow; a 1K file takes about 15 seconds to save. And cassette tape is not very secure; it can be damaged (remember how your car stereo used to eat tapes?) or erased. Care must be taken when recording files because cassette tape data storage can be very sensitive to the "volume" level setting on the cassette recorder. Even with these drawbacks, tape can be a valuable way to store data and programs.

Connect the cassette tape recorder as described in the Model 100 owner's manual.

When the cassette recorder is attached to the Model 100 with the TRS-80 Recorder-to-Computer cable the recorder's manual controls are disabled and control is transferred to the Model 100. To regain control of the tape machine (for rewinding the cassette or whatever) use the command:

MOTOR ON

To turn control of the cassette tape player back to the Model 100 use the command:

MOTOR OFF

CSAVE

To save the current BASIC program file to cassette tape set the cassette recorder to *record*. Use the command:

CSAVE "FLNAME"

to send the file to the recorder. Note that the extension should *not* be included with the filename. Model 100 BASIC is very fussy about this. The Model 100 will start the cassette machine, record FLNAME, and then record the program file on tape. The Model 100 doesn't know if it is recording over an existing file. It is up to you to avoid this, by making sure there's no data on the tape you use.

The ,A option may be added to the command to **CSAVE** the program in ASCII format:

CSAVE "FLNAME",A

CLOAD

To load program files from cassette, rewind the tape. Set the cassette player to play. Then use the command:

CLOAD "FLNAME"

Figure 47 shows the display created by the command **CLOAD.** In this case there were three files on the cassette tape. We asked the Model 100 to **CLOAD** the file name FILE3, using the command:

CLOAD "FILE3"

The Model 100 started the cassette player and began to read the first file on the tape. It did not find FILE3 and displayed the message:

Skip: FILE1

It also skipped over FILE2. When it found FILE3 it displayed the message:

Found:FILE3

The Model 100 loaded FILE3 from the cassette and then displayed the "Ok," BASIC's way of saying that the program has been loaded successfully.

If the **CLOAD** command is followed by a comma and the letter **R:**

CLOAD "FLNAME",**R**

The program will run as soon as it is loaded.

```
Ok
CLOAD "FILE3"
Skip :FILE1
Skip :FILE2
Found:FILE3
Ok
```

Figure 47 CLOAD Screen

CLOAD?

The command **CLOAD?** can be used to verify that the program file in memory is the same as the program file on tape:

CLOAD? "FLNAME"

The question mark is essential. The FLNAME must be the name of the file on tape, not the one in memory.

After the **CLOAD?** command is entered, the tape is searched for FLNAME. If there are no differences, BASIC displays the message, "Ok," as in Figure 48. If there are any differences, the message, "Verify failed" is displayed, as shown in Figure 49. Even if only *one character* is different, "Verify failed" will be displayed. This is a useful command when you want to verify accurate loading of the program.

```
Ok
CLOAD? "FILE2"
Skip :FILE1
Found:FILE2
Ok
```

Figure 48 CLOAD? Screen

```
Ok
CLOAD? "FILE3"
Skip :FILE1
Skip :FILE2
Found:FILE3
Verify failed
Ok
```

Figure 49 CLOAD? Verification Failed

Backing up Files

Now that you know how to save a file on cassette you should know:

THE SIXTH RULE OF PROGRAMMING:

No electronic storage (either magnetic storage such as tape and diskettes, or RAM) is secure.

If the sixth rule is ignored you may lose your programs and data. Even if you don't plan to carry confidential information in the Model 100 you should be concerned about the physical security of your data. The Model 100 stores information electronically. Power failures, dead batteries, magnetic fields, mis-typed KILL commands and a host of other problems can destroy your files. If you ever have to press the RESET button (or worse yet, the Memory Power button), your programs, documents and data files will take a one-way trip to Electromagnetic Never-Never Land.

If you don't have cassette tape copies of *all* your files you could be courting disaster. If you have all your programs and files on cassette tape, you may be able to reload the files from tape to the Model 100.

Backups on the cassette tape should be made whenever you have something that is worth saving—everything that you don't want to recreate from scratch. Make two copies on tape or better yet, on two different tapes. If the first copy won't reload, you can try to reload the second copy. Data files and documents should be backed up at the end of each work day. The cassette should be labeled with the contents and the date. This may seem like a lot of work, but you would not like to lose an important data file, such as all your customer names and addresses. If you travel with the Model 100 you should have a cassette recorder and backup tapes of all important files that you will need during the trip.

Hard Copies of Programs, Data Files, and Documents

Always make hard copies of programs, data files, and documents on the printer (if you have a printer). If you spend hours writing a BASIC program, it's certainly worthwhile to spend a few minutes printing it out (using LLIST). If a program takes more than one day to write, then save the unfinished program on tape at the end of each day and print a listing of the unfinished program.

To wrap up this section of the chapter, before we get to data files, here are some key points to remember when using the cassette machine to save or load BASIC program files. Some of these points

should also be kept in mind when cassette tape is used for data file storage.

- Always use leaderless, certified, data quality tape available at computer retail outlets.
- Follow the sixth rule. Cassette tape is *not* permanent. If the program is important, CSAVE it more than once. You can number consecutive CSAVEs. For example, to save a mailing list program use MAIL1, MAIL2, MAIL3, etc. You can also use separate tapes.
- More about the sixth rule: Programs are stored on tape magnetically. Offices have many electric appliances that produce magnetic fields, such as typewriters, radios, and power cords. Even the loud speaker on the cassette player has a strong magnet in it. Placing the cassette near these devices can cause them to be erased, completely or partially.
- The volume adjustment on some tape machines can be critical. Take time to experiment by creating practice files and recording them onto tape at different volumes
- Write down the filenames of the programs that you save on the cassette label. If you ever have to find out what files are stored on a cassette tape enter this command, exactly as you see it here:

```
CLOAD "XXX"
```

XXX is a dummy filename that you know is not on the tape. BASIC will read the length of the tape and tell you what it contains by displaying:

```
Skip: FILE1
Skip: FILE2
Skip: FILE3
```

and so on until it reaches the end. When the end of the tape is reached, and all the files have been listed, press the SHIFT and BREAK keys to cancel the command.

How're you doing? If you've read straight through the first part of this chapter now might be good time to take a break. The rest

of the chapter really picks up speed as it enters the new territory of data files.

Creating Data Files

In addition to program files there are two other types of files displayed by the main menu. These files have the .DO extension. (See Figure 46.) They are document files and data files. Document files are created by the TEXT word processing program. They store text information (letter, memos, and so on) as ASCII characters. Data files also store information in ASCII characters, but the information is not text. It is data that is to be used by a computer program.

Two examples of data files are ADRS.DO and NOTE.DO. The ADRS.DO file is the data file used by the ADDRSS and TELCOM programs. The NOTE.DO file is the data file used by the SCHEDL program. They are both created by the Model 100 user with the TEXT word processor but they are not documents in the usual sense. They are lists of data that will be used by ADDRSS and TELCOM computer programs.

In this part of the chapter we'll learn how to create data files and the BASIC programs that use them. The first sample program concentrates on the mechanics of creating and using a data file. Successive programs demonstrate the use of files in more complex applications, such as order entry and the mailing list programs.

Model 100 BASIC stores data in sequential data files. This mouthful of jargon means that Model 100 BASIC stores data in sequence, one data item after the next. Four names would be stored in a sequential data file like this:

Washington , Franklin , Hancock , Madison

To find any name in a sequential file BASIC must start with the first name in the file and read every name until it finds the name it's looking for. For instance, to find the name Hancock, BASIC would have to read Washington, Franklin, *and* Hancock before it found Hancock. This is a bit redundant, but as you'll see so is the way BASIC uses sequential files. For example, if BASIC wanted to find the next name, Madison, BASIC would have to start

over again with Washington and read through the list until it read Madison.

Writing programs with sequential data files involves new commands and concepts. For this reason the first program demonstrates the fundamental operations of data file creation and use. It accomplishes eight things that can be done by any program that uses sequential data files to store and retrieve data. The program:

- Gets data from the user.
- OPENs a sequential data file for *storage* of data.
- Stores the data in the file.
- CLOSEs the data file.
- OPENs the data file again for *retrieval* of data.
- Retrieves the data.
- CLOSEs the data file.
- Uses the retrieved data items for calculation or output.

This program is shown in Figure 50. The sections of the program that accomplish the eight steps listed above are indicated by remarks to help you compare the program lines to their function.

Lines 50–60 get two pieces of data from the user; a customer name and the number of the product used by the customer. The input screen is shown in Figure 51. To keep things simple do not enter any commas. (There's a way to set up the program to create sequential files that allow you to use commas in names such as ABC, INC., but it's not included in this book. For one thing, it's more complicated. Since you're writing programs for yourself why complicate your life for the sake of a few commas?)

For example ELECTRONICS INC should not be entered as ELECTRONICS, INC. because BASIC uses the comma to separate pieces of data in the sequential file.

```
10 'PROGRAM ILLUSTRATES USE OF SEQUENTIAL
DATA FILE
20 'STEP 1:
30 'GET DATA FROM USER
40 CLS
50 LINE INPUT "ENTER NAME...";NM$
60 INPUT "ENTER PRODUCT CODE...";PC
70 'STEP 2:
80 'OPEN SEQUENTIAL FILE FOR STORAGE OF DATA
90 OPEN "RAM:DATA.DO" FOR APPEND AS 1
100 'STEP 3:
110 'STORE DATA IN FILE
120 PRINT #1, NM$; ","; PC
130 'STEP 4
140 'CLOSE DATA FILE
150 CLOSE 1
160 CLS :LINE INPUT "PRESS 'ENTER' TO
RETRIEVE DATA...";X$
170 'STEP 5
180 'OPEN DATA FILE FOR RETRIEVAL OF DATA
190 OPEN "RAM:DATA.DO" FOR INPUT AS 1
200 'STEP 6
210 'RETRIEVE DATA FROM THE FILE
220 INPUT #1, NM$, PC
230 'STEP 7
240 'CLOSE DATA FILE
250 CLOSE 1
260 'STEP 8
270 'USE RETRIEVED DATA IN CALCULATION OR
OUTPUT
280 CLS
290 PRINT "HERE IS YOUR DATA..."
300 PRINT:PRINT "THE CUSTOMER - "; NM$
310 PRINT "USES PRODUCT NUMBER";PC
320 END
```

Figure 50 Creating and Using a Sequential Data File

```
ENTER NAME...JONES CORP
ENTER PRODUCT CODE...? 357
```

Figure 51 Data File Input

OPEN Command, for Data Storage

A program has to open a data file whenever data is added or retrieved.

Line 90 opens the data file for storage of data with the **OPEN** command. The command to open takes the form:

OPEN "RAM:FLNAME.DO" **FOR APPEND AS** *file number*

When you open data files in your own programs they'll need two things: a filename and a file number. We're familiar with filenames. File numbers are described below. FLNAME.DO can be any valid filename but it must end in the DO extension, such as DATA.DO in line 90:

```
90 OPEN "RAM:DATA.DO" FOR APPEND AS 1
```

The key words, **FOR APPEND AS** *file number,* indicate that the file is being opened to APPEND, or add, data to the file. If the file is empty APPEND simply starts adding data to the beginning.

We said that when you open a data file you must specify two things: a filename and a file number. BASIC needs that file number because programs can use more than one file and the *file number* is a number that BASIC uses to keep the files separate while the program is running. We'll see how to use two files later.

Since there is only one file in this program the file number is 1. If there were more than this one file in the program they would be numbered 2, 3, etc.

Putting all these parts together gives us the program line:

```
90 OPEN "RAM:DATA.DO" FOR APPEND AS 1
```

Line 90 opens the data file **FOR APPEND**. That means that immediately after the program gets to line 90, the file DATA.DO is opened for storage of information. But that's all; **FOR APPEND** means for storage *only*. The program cannot retrieve data from a file while it is opened for storage of data. We'll see how to open the file for data retrieval in a little bit.

PRINT # Command

The **PRINT #** command is used to send data to the data file.

Line 120 stores the data with the statement:

```
120 PRINT #1 , NM$;",";PC
```

Let's look at line 120. **PRINT #1** causes the data to be sent to file number 1, the open data file. The comma after the file number is required, and is followed by the names of the variables whose values are to be stored. Notice that both string data and numerical data can be sent to the file in one **PRINT #** statement.

There is one more important feature of line 120. The two data items must be stored in the file separated by a comma. The programmer must provide the comma, as shown in line 120 below:

```
120 PRINT #1 , NM$;",";
```

The "," sends a comma to the file between the customer name and the product code. If you don't include the "," BASIC won't know where one piece of information ends and the next begins. Without commas, BASIC would see each line as one huge single piece of information.

After execution of line 120, the file DATA.DO contains the following data in ASCII form:

JONES CORP, 357

Compare this with the input provided by the user in Figure 51.

CLOSE Command

After data is sent to a data file it must be closed. Line 150 closes the file with the command:

```
150 CLOSE 1
```

The file number used in the **CLOSE** command must be the same number that was used to open the file.

OPEN Command, For Data Retrieval

After the file is closed it must be reopened to retrieve data, as shown in line 190.

```
190 OPEN "RAM:DATA.DO" FOR INPUT AS 1
```

Note that the only difference between line 90 and line 190 is the key words, **FOR INPUT AS. FOR INPUT AS** indicates that the file is being opened for data retrieval. Maybe you think this is a lot of trouble to go through just to store and retrieve the words, JONES CORP, 357.

You're right, it is. But this is the way BASIC does it. You can't add data to a file opened for retrieval. You can't retrieve data from a file opened for storage. When you write your own programs you are the only one who can keep track of which files are open for which purpose. This is so important that it's:

THE SEVENTH RULE OF PROGRAMMING:

You can only add data to a file opened FOR APPEND. You can only retrieve data from a file opened FOR INPUT.

INPUT # Statement

The **INPUT # statement** is used to read data from a data file.

Line 220 uses the **INPUT # statement** to retrieve data for use by the program. It follows the form:

INPUT # *file number, variable(s)*

The *file number* is 1. The *variables* we want are NM$, and PC. So the complete **INPUT #** statement looks like this:

```
220 INPUT #1 , NM$ , PC
```

The file must be closed once more and line 250 does this.

Lines 280–310 use the data retrieved from DATA.DO in the output section of the program and display it on the screen, as shown in Figure 52.

```
HERE IS YOUR DATA...

THE CUSTOMER - JONES CORP
USES PRODUCT NUMBER 357
Ok
```

Figure 52 Data File Output

OK. We know you're not going to go to all this trouble to store two pieces of information. This program showed you the mechanics of opening and closing a data file. It showed the importance of the seventh rule: keeping track of storage and retrieval.

But you might write a program that stores sets of related data in files using arrays or data entry of some kind. And that's exactly what we're going to do in the program examples that follow.

A Customer/Product Data Program

This program is fairly spartan. There are almost none of the user-friendly enhancements that this book advocates. The program illustrated in Figure 53 combines the file-handling commands learned above with user-friendly features such as a program menu. It demonstrates the BASIC programming techniques used to store a set of related data in a sequential file—company names with the product purchased by that company. Then you may retrieve the entire set of data to print a customer list. Or you may retrieve any one piece of data by supplying the company name. You're the boss.

```
10 'CUSTOMER DATA PROGRAM USES SEQUENTIAL
FILE
20 'TO STORE COMPANY NAME AND PRODUCT
PURCHASED
30 'AND LAST DATE OF PURCHASE
40 '---------------CREATE MENU
50 CLS:PRINT "   CUSTOMER DATA PROGRAM"
60 PRINT
70 PRINT "1-ADD TO COMPANY LIST"
80 PRINT "2-RETRIEVE COMPANY DATA"
90 PRINT "3-PRINT CUSTOMER LIST"
100 PRINT "4-END PROGRAM"
110 PRINT
120 INPUT "ENTER NUMBER OF YOUR CHOICE..."
;N
130 ON N GOTO 160,220,340,140
140 CLS:PRINT "PRESS 'F4' TO RERUN":END
150 '--------------OPEN DATA FILE, STORE
CUSTOMER DATA
160 OPEN "RAM:CDATA.DO" FOR APPEND AS 1
170 CLS:LINE INPUT "ENTER COMPANY NAME..."
;CN$
180 LINE INPUT "ENTER PRODUCT NAME...";PN$
190 LINE INPUT "ENTER LAST PURCHASE DATE
...";PD$
200 PRINT #1,CN$;",";PN$;",";PD$
```

Figure 53 continues

```
210 CLOSE 1: GOTO 50
220 '--------------OPEN DATA FILE, RETRIEVE
COMPANY DATA
230 OPEN "RAM:CDATA.DO" FOR INPUT AS 1
240 CLS:PRINT
250 LINE INPUT "ENTER COMPANY NAME...";X$
260 IF EOF(1) THEN GOTO 310
270 INPUT #1, CN$,PN$,PD$
280 IF LEFT$(X$,3)<>LEFT$(CN$,3) THEN
GOTO 260
290 PRINT:PRINT "COMPANY- ";CN$:PRINT
"PRODUCT- ";PN$:PRINT "DATE-     ";PD$
300 PRINT:LINE INPUT "PRESS 'ENTER' TO
CONTINUE...";Y$: GOTO 320
310 PRINT X$;:LINE INPUT " NOT FOUND, PRESS
'ENTER'..." XX$
320 CLOSE 1: GOTO 50
330 '--------------PRINT CUSTOMER LIST
340 OPEN "RAM:CDATA.DO" FOR INPUT AS 1
350 LPRINT "COMPANY";TAB(25); "PRODUCT";
TAB(43); "DATE OF LAST PURCHASE"
360 LPRINT " ------------------------------"
370 IF EOF(1) THEN GOTO 410
380 INPUT #1, CN$,PN$,PD$
390 LPRINT CN$;TAB(25);PN$;TAB(43); PD$
400 GOTO 370
410 CLOSE 1: GOTO 50
```

Figure 53 Storing Customer Data in a Sequential Data File

Let's see how the BASIC commands in this program accomplish these things. Some of the commands are the same ones used in the previous program, with added refinements. We'll concentrate on the new features.

Lines 50–120 create the menu shown in Figure 54 and display it on the screen. The user may select one of four options:

```
1-ADD TO COMPANY LIST
2-RETRIEVE COMPANY LIST
```

```
3-PRINT CUSTOMER LIST
4-END PROGRAM
```

```
   CUSTOMER DATA PROGRAM

1-ADD TO COMPANY LIST
2-RETRIEVE COMPANY DATA
3-PRINT CUSTOMER LIST
4-END PROGRAM

ENTER NUMBER OF YOUR CHOICE...? 1
```

Figure 54 Program Menu

Program flow after the menu is controlled by the ON...GOTO command in line 130, depending on the selection made by the user. If the user selects 1, ADD TO COMPANY LIST, the program jumps to line 160. Line 160 uses the OPEN command:

```
160 OPEN "RAM:CDATA.DO" FOR APPEND AS 1
```

This command OPENs the file CDATA.DO for storage of data.

```
ENTER COMPANY NAME...ELECTRONICS CORP.
ENTER PRODUCT NAME...MODEL 100
ENTER LAST PURCHASE DATE...06/24/83
```

Figure 55 Data Entry Screen

Lines 170–190 use LINE INPUT statements to get data from the user, as shown in Figure 55. The values are assigned to three variables:

CN$ - Company name
PN$ - Product name
PD$ - Last date of purchase

When the user has entered all the data, the command in line 200 sends it to the data file. Note the syntax of line 200 carefully.

```
200 PRINT #1, CN$; ",";PN$; ",";PD$
```

First the string for company name, CN$, is sent to the file. Then a comma is sent to the file to separate this first item of data from the next. Then the string for product name, PN$, is sent to the file, followed by another comma. The last string, PD$, does not need to be followed by a comma because it is the last data item on the line. It will be followed by a carriage return supplied by BASIC. Remember that. You supply the commas in the PRINT # statement. BASIC supplies the carriage return at the end of the PRINT # statement. Programmers have to keep track of the smallest details.

Line 210 closes the file and the program jumps back to line 50 where the menu is displayed again.

Adding Data in Sequential Files

Adding data to a sequential file is like adding box cars to a train. Just as the last car added is coupled to the end of the train, the last data items are added to the end of a sequential file. Figure 56 shows what the file, CDATA.DO, would look like after four sets of data have been added. Notice that each set of data is on one line of the file and each data item is separated by a comma. BASIC uses these commas to distinguish among data items when retrieving data from the file.

If you type this program into the Model 100 and run it, the CDATA.DO file will be created. When the program ENDs, press key F8 to go to the Model 100 main menu. You will see that the

file CDATA.DO is displayed in the menu. Place the cursor over CDATA.DO and press ENTER. This brings the contents of CDATA.DO to the screen of the Model 100. It should look similar to Figure 56, but the data file will contain the data that *you* entered. Try it.

```
ELECTRONICS CORP., MODEL 100,06/24/83
SERVICE CO.,MODEM ,01/01/82
MANUFACTURING CO.,PRINTER,04/15/84
RADIO STATION WXYZ,SOFTWARE,11/09/86
```

Figure 56 ASCII DATA File

Once the data has been added to the file it can be used in other sections of the program. From the program menu, Figure 54, the user can select 3-PRINT CUSTOMER LIST. This sends control to line 340, where the file is opened for data retrieval to get the data from the file and send it to the printer. Line 340 handles this in a familiar fashion:

```
340 OPEN "RAM:CDATA.DO" FOR INPUT AS 1
```

Closing a File

In some ways BASIC is great. It does all sorts of stuff for us that would drive us crazy if we had to think about it. But when it comes to data files, BASIC can be very stupid. BASIC reads the data in a sequential file from beginning to end, but BASIC doesn't even know to watch out for the end of the data file unless you tell it to.

If you want to retrieve every item of data in the file you have to start at the beginning and retrieve the data, item by item. When the program reaches the end of the file it has to stop retrieving data. This may seem obvious to you but it is not obvious to BASIC. You have to tell it. This is done with the **EOF** statement:

EOF *(file number)*

Remember that the *file number* is the number (1 in this case) that was used in the OPEN statement. The *file number* is the number of the OPEN file being read. In line 370 it's used in an IF...THEN statement:

```
370 IF EOF(1) THEN GOTO 410
```

The CDATA.DO file was opened as file number 1, so the *file number* is 1. The statement in line 370 works this way: Each time BASIC gets to line 370, BASIC checks to see if it has reached the end of the file. If the end of the file **(EOF)** has been reached, the program jumps to line 410 and the file is closed. If the **EOF** has *not* been reached, control falls through to line 380, and three items of data are INPUT from the file.

```
380 INPUT #1, CN$,PN$,PD$
```

Line 390 sends this data to the printer. Line 400 jumps back to line 370 and tests for **EOF** again. The loop of lines 370–400 are executed until the **EOF** is reached. The resulting customer listing is shown in Figure 57.

```
COMPANY                  PRODUCT    DATE OF
                                    LAST PURCHASE

---------------------------------
ELECTRONICS CORP         MODEL100   06/24/83
SERVICE CO.              MODEM      01/01/82
MANUFACTURING CO.        PRINTER    04/15/84
RADIO STATION WXYZ       SOFTWARE   11/09/86
```

Figure 57 Listing

This section of the program (lines 330–410) retrieves the entire file and prints a list. But what if you need to retrieve only one set of data? What if you know the company name and need to find out the product it buys from you and the last date of purchase? What if you don't even know the correct spelling of the company name? Can BASIC search through the file and retrieve only the data we need? You bet it can. BASIC can search through a data file and pick out one piece of the data that you need. The routine in lines 230–320 shows how this is done.

Searching for Specific Data in Sequential Files

Lines 220–320 search through the data file to find that one set of information that you desire. This routine (lines 220–230) contains a new twist on the use of data files, so follow carefully.

Line 230 OPENs the file CDATA.DO for data retrieval. Line 250 asks the user for the name of the company about which information is required. The string entered by the user is assigned to the variable X$, at the end of line 250. Figure 58 shows the data retrieval screen created by this routine. The user has entered MANUFAC, only part of the company name. Can the program still find the needed data? Let's see.

```
ENTER COMPANY NAME. . .MANUFAC

COMPANY- MANUFACTURING CO.
PRODUCT- PRINTER
DATE-      04/15/84

PRESS 'ENTER' TO CONTINUE...
```

Figure 58 DATA Search Screen

Before reading the first piece of data, line 260 checks for EOF. If the EOF has been reached before finding the required data, it is not in the file. This is true no matter how many pieces of data are in the file. Therefore, the program checks for EOF before it reads each piece of data. If line 260 does find the EOF, control jumps

to line 310, where the user is informed that the company name was NOT FOUND. Too bad.

If the EOF has *not* been reached, the program continues on to line 270:

```
270 INPUT #1, CN$, PN$, PD$
```

Line 270 reads three items of data from the file, CN$, PN$, and PD$.

Remember that the user only entered part of the company name "MANUFAC." Line 280 compensates for this with the LEFT$ function from Chapter 2. It compares only the 3 left characters of X$ with the three left characters of the company name, CN$. If X$ and CN$ are not equal the program jumps back up to line 260 to read the next item of data.

To find any name in a sequential file, BASIC must start with the first name in the file and read every name until it finds the name it's looking for.

In the program in Figure 53 MANUFAC is being sought. If the company name that was retrieved doesn't match MANUFAC, line 280 sends the program back to line 260 to see if the next piece of data in the file is the name we're looking for. Yes, this is complicated. But it follows a simple pattern:

- Check for EOF. If the program reaches the EOF before finding your data then your data just isn't in the file.
- If EOF has not been reached, the program reads the next piece of data from the file.
- Compare the data you just read from the file with the data that it wants.
- When the program finds the data it wants, use it.

Lines 220–320 do this, and when the data required is located line 290 sends it to the screen—as shown in Figure 58.

The data file created in this example was small. But when you are creating a data file you can keep adding data until you run out of Model 100 memory.

Creating Data Files with the TEXT Word Processor

It's not necessary to include a data entry routine in every program that uses a data file. In fact, on the Model 100 it is very easy to create a data file using the TEXT word processor. Because data is stored in data files as ASCII characters, the data can be typed in as well as sent to the file from a program. TEXT can also be used to edit data files. Lines of data can be deleted. New lines of data can be added anywhere, not just at the end of the file. All the features of the TEXT word processor are available while you enter data. Individual data items, such as customer names, can be located with the TEXT *Find* function and changed.

Let's say you want to change the data file shown in Figure 56.

```
ELECTRONICS CORP., MODEL 100,
   06/24/83
SERVICE CO.,MODEM,01/01/82
MANUFACTURING CO.,   PRINTER, 04/15/84
RADIO STATION WXYZ,   SOFTWARE,
   11/09/86
```

Perhaps your customer Service Co. has gone out of business. You can delete their name from the data file by following the steps described below.

The filename of the data file is CDATA.DO. When the screen is displaying the Model 100 main menu, place the cursor over CDATA.DO and press ENTER. This brings the contents of the file to the screen of the Model 100. It should look like the example above.

Use the cursor movement keys to position the cursor at the right-hand end of the SERVICE CO. line. Then press the DEL BKSP key until the entire line is deleted. The file will then look like this:

```
ELECTRONICS CORP.,MODEL 100  06/24/83
MANUFACTURING CO.,   PRINTER, 04/15/84
RADIO STATION WXYZ,   SOFTWARE, 11/09/86
```

But that's not all you can do with TEXT. The next section shows how to create a data file with TEXT and use it in a program. The data files (NOTE.DO and ADRS.DO) used with SCHEDL, ADDRSS and TELCOM have been created with TEXT.

The program in Figure 59 is the Mailing List program from Figure 32 revised to use the ADRS.DO file as a data file.

```
10 'MAILING LIST PROGRAM USES THE DATA
FROM
20 'ADRS.DO TO PRINT MAILING LABELS
30 CLS
40 PRINT "              MAILING LIST
PROGRAM"
50 BEEP
60 PRINT:PRINT "PUT MAILING LABELS IN
PRINTER"
70 PRINT:LINE INPUT "PRESS 'ENTER' TO
CONTINUE...";X$
80 '--------------OPEN DATA FILE
90 OPEN "RAM:ADRS.DO" FOR INPUT AS 1
100 IF EOF(1) THEN GOTO 230
110 '--------------GET DATA FROM FILE
120 INPUT #1, N$,CN$,ST$,CS$,ZP%, TN$
130 '--------------PRINT THE INFORMATION
ON LABEL
140 LPRINT N$
150 LPRINT CN$
160 LPRINT ST$
170 LPRINT CS$,"   ";
180 LPRINT ZP%
190 LPRINT
200 LPRINT
210 '--------------REPEAT UNTIL
ALL LABELS PRINTED
220 GOTO 100
230 CLOSE 1
240 CLS
250 PRINT "              MAILING LIST
```

Figure 59 continues

```
PROGRAM"
260 PRINT:PRINT "PRESS 'F4' TO RERUN"
270 END
```

Figure 59 Mailing List Program Using ADRS.DO File

Before the mailing list program can be used with ADRS.DO, an ADRS.DO data file must be created. This file will also have to be compatible with the other programs that already use it—the Model 100's ADDRSS and TELCOM programs. To create this data file perform the following three steps:

1. From the main menu, place the cursor over TEXT. Press ENTER.

2. When the message "File to edit?" appears on the screen, type:

   ADRS

   and press ENTER. This just gives the Model 100 a filename for the file you are about to create. The TEXT program automatically supplies the DO extension.

3. The third step requires a short explanation. The Mailing List program recognized the following variables. It expects to find the data items listed in the following order:

N$	Customer name
CN$	Company name
ST$	Street and number
CS$	City and Street
ZP%	Zip code
TN$	Telephone number

Because the program expects to find the data items listed in this order, you've got to create a data file that contains the data arranged in each line of the file in the same order. The telephone number, included for use by TELCOM, should be the last item on each line. A sample line of data in ADRS.DO is shown below:

```
Mr. S. Smith, A.B.C. Inc., 123 Main St.,
Anytown NY, 12345,:21255551212:
```

Separate each data item by a comma. Do not include any commas in the file except those that separate data items. When the data you are typing reaches the right-hand side of the screen, it will "wrap around" automatically. Terminate each line of the data file by pressing ENTER. There should be only one carriage return for each line of data.

```
Mr. S. Smith,A.B.C. Inc.,123 Main
St.,Anytown NY,12345,:2125551212:
Ms. J. Jones,Electro Corp.,111 Center
St.,Valley NY,12345,:2125551221:
Mr. N. Gineer,Computer Co.,21 Inter Ave.,
New City NY,12345, :2125551234:
Ms. A. Name,X.Y.Z. Inc.,11 Exec Way,
Central NY,12345,:2125554321:
```

Figure 60 ADRS.DO File

The sample data file used with the Mailing List program is shown in Figure 60. Notice how much it resembles the data statements of the previous Mailing List program, Figure 32. It is, after all, just another place to store data. (But the data in a data file can be used by more than one program. And as we've seen, data can be added to a data file while the program is running. You can't do that with DATA statements.)

After the introductory display is created by lines 30–70, line 90 opens the ADRS.DO file in the usual way:

```
90 OPEN "RAM:ADRS.DO" FOR INPUT AS 1
```

Line 100 checks for EOF. If the EOF is found, the program jumps to line 230 and the file is closed. But if the EOF is not found, control falls through to line 120. In line 120 an INPUT # statement is used to retrieve six items of data from the file. Then lines 140–200 print the mailing label.

Note that it is not necessary to use all the data items that are INPUT from the data file. The telephone number is not printed on the label. But it is necessary for line 120 to INPUT all the data

items in a sequential data file or the program will lose track of its place in the data.

Line 220 sends the program back to line 100, the top of the loop, to check for EOF and get more data. This loop of lines 100–220 continues until the EOF is reached. Then the file is closed in line 230. The output—mailing labels—should look like Figure 61.

```
Mr. S. Smith
A.B.C. Inc.
123 Main St.
Anytown NY    12345

Ms. J. Jones
Electro Corp.
111 Center St.
Valley NY    12345

Mr. N. Gineer
Computer Co.
21 Inter Ave.
New City NY    12345

Ms. A. Name
X.Y.Z. Inc.
11 Exec Way
Central NY    12345
```

Figure 61 Mailing Labels Output

Storing Program Results in Data Files: The Order Entry Program

Up to this point data files have been the *source* of data for the program to use. The *output* of program results has been sent to the screen or to the printer.

But data files are very versatile. Just as it is possible for more than one program to use the same data file, it is also possible for one program to use more than one data file. It is also possible for

a program to create a new data file for use by another program or an output routine. It is possible to send the results of calculations or other output from the program to a data file for storage. With the Model 100's telecommunications capabilities (covered in the next chapter) the new data file can even be sent over telephone lines to be used by another computer.

The Model 100 could be used to write orders in a customer's office. The order information could be saved in a data file and this new data file used to print out the order when the salesperson returns to his or her own office. The data in the new file could also be sent to a home office computer for order fulfillment or other purposes. No hard copy needs to be prepared until the order confirmation is sent to the customer.

In addition, a file could be provided that would supply the cost of each item that the salesperson is likely to sell. This file could be updated as required using TEXT. The program itself would not have to be changed.

The program in Figure 62, an Order Entry program, does these things. The products used in this example are books. It uses the data from one data file, PRICE.DO, to calculate the cost of an order of books. Then it stores the order information in a new data file. The new data file is used by an optional routine in the same program to print the order confirmation when a printer is available. Obviously, you don't want to carry a printer into a customer's office. But with the order information stored in a data file you can print out an order confirmation when you get back to your office. Here's how the program works:

```
10 'ORDER ENTRY PROGRAM RETRIEVES PRICING
INFORMATION
20 'FROM ONE DATA FILE CALCULATES TOTAL
COST OF ORDER
30 'AND STORES ORDER INFO IN ANOTHER DATA
FILE
40 '---------------PROVIDE FOR MORE THAN
ONE FILE
50 MAXFILES = 2
60 '---------------CREATE MENU
```


Figure 62 continues

```
70 CLS:PRINT "   ORDER ENTRY PROGRAM"
80 PRINT
90 PRINT "1-ENTER CUSTOMER ORDER"
100 PRINT "2-PRINT ORDER FROM DATA FILE"
110 PRINT "3-END PROGRAM"
120 PRINT:INPUT "   ENTER NUMBER OF YOUR
CHOICE...";X
130 CLS
140 ON X GOTO 160,560,880
150 '---------------GET ORDER INFO FROM
USER
160 CLS:PRINT "      ENTER DATA REQUESTED
AFTER PROMPT:"
170 PRINT "        (DO NOT ENTER ANY
'COMMAS'!)"
180 PRINT
190 LINE INPUT "CUSTOMER NAME....";NM$
200 LINE INPUT "COMPANY NAME.....";CN$
210 LINE INPUT "STREET ADDRESS...";ST$
220 LINE INPUT "CITY AND STATE ...";CS$
230 LINE INPUT "ZIP CODE.........";ZP$
240 CLS
250 '----------------ASK USER FOR NAME OF
FILE TO STORE DATA
260 FILES
270 PRINT
280 PRINT "ENTER NAME OF DATA FILE FOR THIS
ORDER"
290 LINE INPUT "EXAMPLE:
'FLNAME.EX'....";FN$
300 PRINT
310 "--------------OPEN FILE, STORE
NAME/ADDRESS DATA
320 OPEN FN$ FOR APPEND AS 1
330 PRINT #1, NM$;",";CN$;",";ST$;
",";CS$;",";ZP$
340 '--------------GET ORDER DATA FROM
USER
```

Figure 62 continues

```
350 CLS
360 LINE INPUT "PRODUCT NAME?...";PN$
370 INPUT "QUANTITY.....";Q
380 '--------------OPEN PRICE FILE AND
SEARCH FOR PRICE DATA
390 OPEN "RAM:PRICE.DO" FOR INPUT AS 2
400     IF EOF(2) THEN GOTO 430
410     INPUT #2, PX$, PR
420     IF LEFT$(PX$,5)<>LEFT$ (PN$,5)
        THEN GOTO 400
430 CLOSE 2
440 '--------------CALCULATE ITEM TOTAL
450 PT=Q*PR
460 '--------------STORE PRICE DATA
470 PRINT #1, PX$;",";PR;",";Q; ",";PT
480 '--------------MORE DATA?
490 LINE INPUT "MORE DATA? (Y/N)...";X$
500 PRINT
510 IF X$="Y" OR X$="y" THEN GOTO 360
520 '--------------NO MORE DATA CLOSE
DATA FILE
530 CLOSE 1
540 GOTO 60
550 '--------------PRINT ORDER
560 FILES
570 PRINT
580 LINE INPUT "FILE TO PRINT?...";FN$
590 OPEN FN$ FOR INPUT AS 1
600 IF EOF(1) THEN GOTO 820
610 INPUT #1,NM$,CN$,ST$,CS$,ZP$
620 LPRINT "ORDER CONFIRMATION...
WHOLESALE BOOKS"
630 LPRINT "123 MAIN STREET, ANYTOWN,
NY      PHONE:555-1234"
640 LN$="------------------------------"
    ----------------------------------
650 LPRINT LN$
660 LPRINT "CUSTOMER:",,,,"DATE:"
```

Figure 62 continues

```
670 LPRINT NM$,,,,DATE$
680 LPRINT CN$
690 LPRINT ST$
700 LPRINT CS$;"    ";ZP$
710 LPRINT LN$
720 LPRINT "ITEM";TAB(22);"$/UNIT",
"QUANT.","COST"
730 LPRINT LN$
740     IF EOF(1) THEN GOTO 820
750     INPUT #1, PN$,PR,Q,PT
760     LPRINT PN$;TAB(20);
770     LPRINT USING "$$ ###.##"; PR,
780     LPRINT ,Q,
790     LPRINT USING "$$ ###.##";PT
800     TL=TL+PT
810     GOTO 740
820 LPRINT LN$
830 LPRINT "ORDER TOTAL"; TAB(20),,,
840 LPRINT USING "$$ ###.##";TL
850 LPRINT:LPRINT "THANK YOU FOR YOUR
ORDER"
860 CLOSE 1
870 GOTO 60
880 CLS:PRINT "PRESS 'F4' TO RERUN..."
890 END
```

Figure 62 Program Showing Two Files Open at One Time

The MAXFILES Command

During execution of the Order Entry program two files will be open at one time: the PRICE.DO file that contains the prices, and the MAIN.DO file that contains the order information. If you plan on having more than one file open you have to tell the Model 100 how many files to expect. That is because areas of memory must be partitioned off for transfer of data to and from the file. (These areas of memory are called data buffers and are created automatically by BASIC.) Line 50 takes care of this with the **MAXFILES** command:

MAXFILES = *number of files*

MAXFILES is a clever acronym for the **MAX**imum number of **FILES** in the program. The *number of files* in this case is 2, because two files are to be open while the program is running. Lines 70–120 display a program menu as shown in Figure 63. The user has three options. He can enter a customer order, print an order confirmation from an existing data file, or end the program. Program flow at this point is controlled by the ON...GOTO statement in line 140.

```
   ORDER ENTRY PROGRAM

1-ENTER CUSTOMER ORDER
2-PRINT ORDER FROM DATA FILE
3-END PROGRAM

   ENTER NUMBER OF YOUR CHOICE...? 1
```

Figure 63 Program Menu

If the user chooses to enter a customer order, the program jumps to line 160. This is the first line of the order entry routine (lines 160–540). This is one of the most complex programs in the book. As we said in Chapter 1, a BASIC program can only do one thing at a time. Therefore, if the program is organized, it can be followed line by line, and the purpose and function of each line analyzed. This is important to keep in mind when writing programs as well as when looking at existing programs.

The order entry routine (Figure 62) accomplishes a lot. It:

- Asks the user for the name and address of the customer.
- Asks the user for the name of the new file that is to be created to store the order data. (To help the user choose a unique filename a directory of files is displayed.)
- OPENs the file and stores the name-and-address data.

- Asks the user to enter the order data: product name and quantity. Note that it does *not* get the price from the user.
- OPENs a data file called PRICE.DO, which contains the product names and unit prices. The program searches for the name of the product and retrieves its price. (At this point two files are open.)
- CLOSEs PRICE.DO
- Uses the price information from PRICE.DO to calculate the price and the quantity of each product ordered.
- Stores the data (product name, price, quantity and cost of the item) in the new data file.
- Asks the user if the customer wants to order something else.
- If there is no more data, the new data file is CLOSED and the program returns to the menu.

Input Variables

There are quite a few variables used in the program:

NM$	Customer name
CN$	Company name
ST$	Street and number
CS$	City and state
ZP$	Zip code
FN$	User's choice of filename for order data file
PN$	Name of product ordered
Q	Quantity ordered
PX$	Product name from PRICE.DO file
PR	Price
PT	Total price for quantity of product ordered

Lines 160–230 create the data entry screen shown in Figure 64. (Note that the user is instructed not to enter any commas.) After the prompt:

```
CITY AND STATE...
```

if the user entered:

```
ANYTOWN, NY
```

the comma would make them appear to BASIC to be two separate data items.

```
      ENTER DATA REQUESTED AFTER PROMPT:
        (DO NOT ENTER ANY 'COMMAS'!)

CUSTOMER NAME....MR. S. SMITH
COMPANY NAME.....MAIN BOOKS
STREET ADDRESS...123 MAIN ST.
CITY AND STATE...ANYTOWN NY
ZIP..............12345
```

Figure 64 DATA Entry Screen

FILES

The user is asked to choose the filename for the new data file. Line 260 uses the command, **FILES,** to display a directory of existing files, as shown in Figure 65. The asterisk shown after the filename ORDDER.BA* indicates that it is the current BASIC program. The *current* BASIC program is whatever BASIC program you are using.

```
MY      .BA    ORDDER.BA*   PRICE .DO
SORT1   .BA    MAIL1 .BA    PRDCT .BA
ADRS    .DO
ENTER NAME OF DATA FILE FOR THIS ORDER
EXAMPLE: 'FLNAME.EX'....MAIN.DO
```

Figure 65 User Names DATA File

Using a Variable As a Filename

Lines 280–290 complete the screen display. The filename provided by the user is MAIN.DO, because the company name is Main Books. The filename is assigned to the string variable, FN$. Line 290 uses a string variable in a way that we haven't yet seen. When the file is opened in line 320, the variable FN$ is used in place of a filename:

```
320 OPEN FN$ FOR APPEND AS 1
```

Why is this? When the program gets to line 290 it doesn't know what the filename is going to be. The user has to supply it. This makes it possible to use a filename that indicates which customer placed the order in the file you are creating. (We used MAIN.DO because the customer is Main Books. But we could have used some other name.) And the only way you can enter the filename while the program is running is in response to a LINE INPUT statement. The file will be opened with the filename that the user supplies, MAIN.DO, as if the entire name were used in line 320.

Line 330 sends the name-and-address data to the new file, separated by commas. The file is left OPEN, as the program gets more information ready to send.

MAIN.DO is file number 1. And because it's open, if we want to open the PRICE.DO data file it must be opened as file number 2, as shown below.

Lines 360–370 get the name and the quantity of the first item ordered. Then line 390 opens the existing file, PRICE.DO. Look carefully at the OPEN statement:

```
390 OPEN "RAM:PRICE.DO" FOR INPUT AS 2
```

The file number is 2, because another file, in this case MAIN.DO, is already open. Similarly, the file number used for the PRICE.DO in the EOF command and the INPUT # statement is 2. It is up to the programmer to keep track of the number of files OPEN. If another file were to be OPENed before any of these files were CLOSEd it would be file number 3.

The contents of PRICE.DO are shown in Figure 66. The name of each product is followed by the price.

```
PRICE.DO, PRICING DATA FILE:

COMPUTER BOOK,12.50
ROMANCE NOVEL,2.50
ADVENTURE NOVEL,2.50
SPY NOVEL,2.75
HORROR NOVEL,2.25
COOK BOOK,5.00
ART BOOK,25.00
```

Figure 66 DATA Files Used by PRICE.DO

Lines 400–420 retrieve the product price.

The INPUT # statement in line 410 retrieves two items of data, PX$ and PR, the product name and the price:

```
410 INPUT #2, PX$, PR
```

In line 420 the product name entered by the user in line 360, PN$, is compared to the product name stored is the PRICE.DO file, PX$. When the names match the file is CLOSEd. The price, PR, will be used to calculate the total cost, PT, in line 450.

The new data file, MAIN.DO, is waiting for the order data. Line 470 sends it: product name, price, quantity, and total price for product. Note that the variable PX$ is used to send the product name to the file. This assures that the correct spelling of the product is sent to the new file, because PX$ contains the product name as it was entered in PRICE.DO. Line 490 asks the user if there is more data. If there is, the program jumps to the top of the data entry routine (line 360) and the process is repeated. If there isn't additional data then MAIN.DO is closed in line 530 and the program returns to the menu.

The order entry screen shown in Figure 67 prompts the user through the steps described above. When the order entry process is completed, MAIN.DO contains the name and address of the customer, plus the data on each book ordered. The contents of

MAIN.DO are shown in Figure 68. The first two lines contain the name-and-address information. The succeeding lines contain the product name, unit price, quantity ordered, and total price for the quantity ordered.

```
PRODUCT NAME?...SPY NOVEL
QUANTITY.........? 12
MORE DATA? (Y/N)...Y

PRODUCT NAME?...COMPUTER
QUANTITY.........? 24
MORE DATA? (Y/N)...N
```

Figure 67 Data Review Screen

```
MAIN.DO, ORDER ENTRY DATA FILE:

MR. S. SMITH,MAIN BOOKS,123 MAIN
ST., ANYTOWN NY,12345
SPY NOVEL, 2.75 , 12 , 33
COMPUTER BOOK, 12.5 , 24 , 300
```

Figure 68 DATA Files Used by MAIN.DO

Printing the Order Confirmation

This program is long and does a lot of new things. The next thing that it does is not new. It opens the MAIN.DO file and prints an order confirmation on a printer.

When the salesperson completes the order entry process, the data file, MAIN.DO, contains all the information needed to print the order confirmation shown in Figure 69. The program contains a routine (lines 560–870) to print the order confirmation. This is how it works:

If the user selects option 2 from the menu, the program jumps to line 560. Lines 560–580 create the display shown in Figure 70.

```
ORDER CONFIRMATION...WHOLESALE BOOKS
123 MAIN STREET,
ANYTOWN, NY     PHONE:555-1234
-------------------------------------
CUSTOMER:                DATE:
MR. S. SMITH             12-16-83
MAIN BOOKS
123 MAIN ST.
ANYTOWN  NY     12345
-------------------------------------
ITEM                $UNIT  QUANT.     COST
-------------------------------------
SPY NOVEL           $2.75     12    $33.00
COMPUTER BOOK      $12.50     24   $300.00
-------------------------------------
ORDER TOTAL                          $333.00

THANK YOU FOR YOUR ORDER
```

Figure 69 Order Confirmation

The user is asked to choose the file to be printed from those displayed. Our choice is MAIN.DO.

```
MY      .BA  ORDDER.BA*  PRICE .DO
SORT1  .BA  MAIN   .DO   MAIL1 .BA
PRDCT  .BA  ADRS   .DO

FILE TO PRINT?...MAIN.DO
```

Figure 70 User Names File to Send to Printer

Line 590 OPENs the file specified by the user. Note that line 590 uses the variable FN$ for the filename. Line 610 INPUT's the

name-and-address values. Lines 620–650 print the heading of the order confirmation. Note that the string variable LN$ is used to print the dotted line separating sections of the order confirmation.

Lines 660–710 print the customer name and address on the confirmation. Note that the DATE$ function is used in line 670 to print the current date on the form:

```
670 LPRINT NM$,,,,DATE$
```

By using the DATE$ function, the program gets the date from the Model 100 built-in calendar. Setting the calendar was explained in Chapter 0.

There is another new feature used in this line, the four commas (,,,,) after the LPRINT command. A semicolon (;) placed after a PRINT or LPRINT statement suppresses the carriage return. A comma placed after such a statement causes a TAB to be inserted at that point. Placing four commas before the DATE$ causes four TABS to be inserted and positions the date at the right hand side of the confirmation. Commas can also be used to line up columns of output. The commas between the headings in line 720 position them at the TAB stops. Then in lines 760–790 commas are used again to position the values under their respective headings.

But let's backtrack. The program still has to get the product order information to the printer. Line 740 checks for EOF. If there is data in the file, line 750 INPUTs it. And the loop of lines 740–810 is executed until the end of the file has been reached.

Notice that each time through the loop line 800 adds the line total for each product ordered, PT, to the total for the entire order, TL:

```
800 TL=TL+PT
```

The value of TL is used in line 840. PRINT USING statements, introduced in Chapter 2, produce the dollars-and-cents format in the second and fourth columns of the order confirmation.

```
840 LPRINT USING "$$#####.##";TL
```

For example, the expression:

```
LPRINT USING "$$####.##"
```

tells BASIC to send the data to the printer in dollars and cents format. All the different PRINT USING formats are given in the Model 100 user's manual.

A Program Using Cassette Files

A salesperson may have to produce a periodic report listing the invoice number, company name, and dollar amount of a year's worth of sales. Let's assume that there are 100 invoices to be recorded. If you are the salesperson and you are writing the program, how could you tell if your Model 100 has enough memory available to store these 100 invoices? That's a good question, and an important one. If you are running a program that you wrote to store information the one thing you don't want to see on this display screen is this:

```
?OM Error
```

This is an "Out of Memory error message." It would mean that your Model 100 just doesn't have any more room to store your data. One limitation of the Model 100 is memory space. There just is not enough of it for some applications. Too bad. How can you get around this? If you have the 8K version your memory allocation might look something like this:

ADRS.DO and NOTE.DO	2K
BASIC program	2K
Document file (memo or letter)	2K
Memory available for data file	2K

The Salesperson's Sales Report Program

What about the information that we want to store? The information to be stored in the file includes the invoice number, company name, and amount of sale for each invoice. It will be stored like this:

101, ABC CO., 2525.50

This is one line of data, and the jargon for one line of data in a data file is a record. Each record will occupy about 23 bytes of RAM, so be sure to count the spaces and the carriage return since every character is stored as one byte. Let's say that you will need an average of 25 bytes to store each line of data in the sequential file. If there are about 2K, or 2048 bytes, of RAM available, then about 80 records will fit. That leaves 20 invoices out in the cold. Of course, this is just an example, but the problem is real. Even if you have a 32K Model 100 you can still run out of memory, depending on how many programs you have written and how much data you have. How can you get around this limitation placed on you by the meager amount of memory in the Model 100? Figure 71, Salesperson's Sales Report program, solves this problem by storing the invoice data on tape and bypassing RAM altogether.

```
10 'PROGRAM WITH DATA ENTRY ROUTINE THAT
20 'STORES DATA IN CASSETTE TAPE DATA FILE
30 '----------------MENU
40 CLS
50 PRINT "SALESPERSON'S SALES REPORT
PROGRAM"
60 PRINT
70 PRINT "1-ENTER DATA & SEND TO TAPE"
80 PRINT "2-PRINT REPORT FROM TAPE FILE"
90 PRINT "3-END PROGRAM"
100 PRINT
110 INPUT "ENTER NUMBER OF YOUR
```

Figure 71 continues

```
CHOICE...";X
120 ON X GOTO 140,430,660
130 '----------------OPEN CASSETTE FILE
140 CLS:BEEP
150 PRINT "PREPARE TAPE PLAYER TO RECEIVE
DATA."
160 LINE INPUT "PRESS 'ENTER' TO
CONTINUE...";X$
170  OPEN "CAS:SDATA.DO" FOR OUTPUT AS 1
180 '---------------DATA ENTRY SCREEN
190 CLS
200 PRINT "ENTER INFO REQUESTED FOR EACH
INVOICE..."
210 PRINT
220 LINE INPUT "INVOICE NUMBER----";IN$
230 LINE INPUT "CUSTOMER NAME-----";CN$
240 INPUT "AMOUNT OF SALE---";AS
250 '--------------DISPLAY DATA FOR
REVIEW
260 CLS
270 PRINT "INVOICE NUMBER--- ";IN$
280 PRINT "CUSTOMER NAME---- ";CN$
290 PRINT "AMOUNT OF SALE--- ";AS
300 PRINT
310 LINE INPUT "IS DATA CORRECT?
(Y/N)...";X$
320 IF X$="Y" OR X$="y" THEN GOTO 360
330 CLS: PRINT "O.K. ENTER DATA AGAIN..."
340 PRINT
350 GOTO 220
360 '--------------SEND DATA TO TAPE
370 PRINT #1, IN$;",";CN$;",";AS;",";
380 '--------------MORE DATA?
390 LINE INPUT "MORE DATA? (Y/N)...";X$
400 IF X$="Y" OR X$="y" THEN GOTO 190
410 CLOSE 1
420 GOTO 40
430 '--------------PRINT FILE AND
```


Figure 71 continues

```
CALCULATE TOTAL SALES
440 CLS:LN$="-------------------------
----------"
450 LPRINT "SALESPERSON'S YEARLY SALES
REPORT
460 LPRINT:LPRINT "INVOICE", "CUSTOMER",
"$SALE"
470 LPRINT LN$
480 BEEP
490 PRINT "PREPARE TAPE PLAYER TO SEND
DATA."
500 LINE INPUT "PRESS 'ENTER' TO
CONTINUE...";X$
510 '--------------OPEN FILE FOR DATA
RETRIEVAL
520 OPEN "CAS:SDATA.DO" FOR INPUT AS 1
530 PRINT:PRINT "PLEASE WAIT. GETTING
DATA."
540     IF EOF(1) THEN GOTO 600
550     INPUT #1, IN$,CN$,AS
560     LPRINT IN$,CN$,
570     LPRINT USING "$$#####,.##";AS
580     TS=TS+AS
590     GOTO 540
600 CLOSE 1
610 LPRINT LN$
620 LPRINT "TOTAL SALES FOR PERIOD",
630 LPRINT USING "$$#####,.##";TS
640 LPRINT:LPRINT "END OF REPORT"
650 GOTO 40
660 CLS :PRINT "PRESS 'F4' TO RERUN":END
```

Figure 71 Program Using Cassette File

The Salesperson's Sales Report program stores data on cassette tape. The program gets data from the user and stores it on tape in a sequential file. Then an output routine gets the data from the file and sends it to the printer. The data file does not use large amounts of RAM because the data is shuttled from the keyboard to tape,

then from the tape to the printer. In fact, the size of a cassette data file is limited only by the amount of data that could fit on one side of a data cassette.

Lines 40–120 create a menu and use the now familiar ON...GOTO command to control program flow based on the user's input. The menu is shown in Figure 72.

```
SALESPERSON'S SALES REPORT PROGRAM

1-ENTER DATA & SEND TO TAPE
2-PRINT REPORT FROM TAPE FILE
3-END PROGRAM

ENTER NUMBER OF YOUR CHOICE...? 1
```

Figure 72 Menu

```
PREPARE TAPE PLAYER TO RECEIVE DATA.
PRESS 'ENTER' TO CONTINUE...
```

Figure 73 User-Friendly Feature

If the user chooses 1-ENTER DATA AND SEND TO TAPE, the display shown in Figure 73 is created. This is a user-friendly feature that reminds the user to ready the cassette player.

Opening Cassette Files

Line 170 opens the cassette file:

```
170 OPEN "CAS:SDATA.DO" FOR OUTPUT AS 1
```

The tape player will start to run as the filename is recorded. Note that the mode specified is FOR OUTPUT. Data can't be APPENDed to the end of a cassette file. A cassette file can only be OPENed once for OUTPUT. *Only* once. Each time a cassette file is opened for OUTPUT, BASIC starts to record data at the beginning of the file. Therefore, all the data must be output to the file before it is closed. If a *cassette* file is opened again, even with the same filename, BASIC starts at the beginning again with whatever data is sent to the file.

```
ENTER INFO REQUESTED FOR EACH INVOICE...

INVOICE NUMBER---- 101
CUSTOMER NAME---- ABC CO.
AMOUNT OF SALE----? 2525.50
```

Figure 74 Data Entry Screen

Lines 190–240 are the familiar data-entry routine from Figure 19, revised for use in this program. Lines 220–240 get the information about one invoice from the user. The data-entry screen is shown in Figure 74. The data is assigned to these variables:

IN$ Invoice number

CN$ Company name

AS Amount of sale

This data is displayed for review by the user in lines 270–290, as shown in Figure 75. The program, in line 310, asks if the data is correct:

```
310 LINE INPUT "IS DATA CORRECT?
(Y/N)...":X$
```

```
INVOICE NUMBER---  101
CUSTOMER NAME----  ABC CO.
AMOUNT OF SALE---2525.5

IS DATA CORRECT? (Y/N)...Y
MORE DATA? (Y/N)...Y
```

Figure 75 Data Review Screen

```
INVOICE NUMBER---1001
CUSTOMER NAME----Q.B.
AMOUNT OF SALE---453.55

IS DATA CORRECT? (Y/N)...N
```

Figure 76 Data Entered Incorrectly

```
O.K. ENTER DATA AGAIN...

INVOICE NUMBER----1001
CUSTOMER NAME-----Q.B.
AMOUNT OF SALE---? 4535.50
```

Figure 77 Corrected Data Entered by User

If, as shown in Figure 76, the user enters anything but Y or y the message:

```
O.K. ENTER DATA AGAIN
```

is displayed, as shown in Figure 77 and line 350 sends the program back to line 220. It is important to display a message such as this to let the user know that he is expected to *reenter* the data.

Data Buffers in BASIC

If the data has been entered correctly, the user responds to the question in line 310 with a Y or y and the program jumps to line 360. Line 370 sends the data to the tape file.

If you enter this program and RUN it you will notice that the tape player does not run every time the program gets to line 370. This is because the data is first sent to a data buffer in memory. A data buffer is a portion of memory that has been automatically set aside by BASIC to hold data until it is sent to a device. (In this case, the tape player.) When the data buffer is full the data is sent to the cassette player. That is why the tape player appears to record information in "spurts."

You can get a feel for how the data buffer functions by running the following program:

```
10 DIM A(100)
20 OPEN "CAS:ARRAY.DO" FOR OUTPUT AS 1
30 FOR X=1 TO 100
40 A(X)=X
50 PRINT #1, A(X);",";
60 NEXT X
70 CLOSE 1
80 END
```

This program saves an array, A(X), with 100 elements in the cassette file, ARRAY.DO. As the program runs you will see the cassette tape player periodically start and stop—start and stop—start and stop. Whenever the data buffer is full, BASIC starts the cassette player and sends the data from the buffer to the cassette file. Then the program proceeds, sending more data to the buffer, then to the tape, and so on until all the data is saved on tape.

Line 380 of the Salesperson's Sales Report program asks the

user if there will be more data to send to the tape file. The loop of lines 190–400 continues to execute until the user has no more data to enter.

```
SALESPERSON'S YEARLY SALES REPORT
INVOICE      CUSTOMER      $ SALE
-----------------------------------
101          ABC CO.        $2,525.50
202          XYZ INC.       $1,500.05
305          APPEX            $250.35
505          ZTEK           $2,001.00
1001         Q.B.           $4,535.50
-----------------------------------
TOTAL SALES FOR PERIOD     $10,812.40

END OF REPORT
```

Figure 78 User-Friendly Feature

```
PREPARE TAPE PLAYER TO SEND DATA.
PRESS 'ENTER' TO CONTINUE...
PLEASE WAIT. GETTING DATA.
```

Figure 79 Sales Report Listing

The routine in lines 430–640 produces the report shown in Figure 78. This routine uses the same algorithm as the previous programs. It OPENs the file for INPUT, checks for EOF, gets the data items, and sends them to the printer. One useful feature is shown in Figure 79. Line 530 displays the message:

```
PLEASE WAIT. GETTING DATA.
```

As we mentioned above, tape files are slow. This message lets the user know that the program is busy retrieving the data.

The loop of lines 540–590 continues until there is no more data left in the file, the file is closed, and the program jumps back to the menu.

More About Model 100 I/O and Devices

As we mentioned in the beginning of the chapter, the programmer's control over I/O on the Model 100 is extensive. Each I/O device can be treated as a separate "file." When you write a program with an OPEN statement:

OPEN "*device*:FLNAME.EX" FOR OUTPUT AS 1

The *device* can be more than just RAM: or CAS:. Model 100 BASIC lets the programmer treat any device as if it were a file. This means programs or data can be sent to the following locations:

If *device* is:	**Location is:**
RAM:	Random access memory
CAS:	Cassette tape
LCD:	Display screen
LPT:	Printer
MDM:	Telecommunications MODEM
COM:	RS 232C Communications Interface

If you type the command:

```
SAVE "LPT:FLNAME.BA"
```

it will produce the same result as typing the command:

```
LLIST
```

The file, FLNAME.BA, will be sent to the device, LPT, the printer.

The command:

```
SAVE "LCD:FLNAME.BA"
```

LCD stands for the Liquid Crystal Display screen and this command has the same effect as:

```
LIST.
```

Similarly, the command:

```
SAVE "CAS:FLNAME.BA"
```

has the same effect as the command:

```
CSAVE "FLNAME".
```

Now that you have finished Chapter 5, you should have a very good feeling for the BASIC programming language. At this point, we've covered all the fundamental things that BASIC can do. You should feel pretty good about that.

Try out your new knowledge by writing programs of your own. The Model 100 and the BASIC language are so versatile that you should soon be writing very useful programs to make you more productive in your work.

Chapter 6

Telecommunications: Computers, the Telephone Lines, and Information

Computers are talking to each other and people are listening. Computers communicate over the telephone lines and the result, called telecommunications, is changing the way we talk to each other, and the way we do business.

More and more business people are plugging their computers into the telephone network. Computer users in small towns and big cities throughout the United States are turning to their display screens for information—everything from the price of a hot stock to the recipe for a hot meal.

The Model 100, with its TELCOM telecommunications software, puts the information universe in your hands. What was once the dream of science fiction writers is as close as your keyboard. You can send and receive customer information, price quotes, technical documents, letters and memos to and from any computer that can be reached by the telephone lines. But that's not all.

Special telecommunications companies have been set up to provide a vast range of information, for a fee, over the phone lines. These companies, such as CompuServe and Dow Jones News/Retrieval, have sophisticated data banks and communications networks that store, organize, and send all this information to your Model 100, wherever there's a telephone.

You can read and send electronic mail, post messages on bulletin boards, and join Special Interest Groups (SIGs). Personal computing SIGs provide free software, computer art, challenging games, programming languages, and new product reviews. Through Model 100 SIGs you can get solutions to problems, receive free computer programs, and share information and experience with other interested users.

But before you punch in the number of your local bulletin board, you have to know the ins and outs of TELCOM, Model 100's versatile telecommunications software. The following sections describe its use.

Communications Parameters

When microcomputers talk to each other, they speak in ASCII code. The ASCII code, described in Chapter 2, gives computers a code that represents each character of the alphabet. Each character of information shared by the computers is conveyed as an ASCII character code. When you send a memo from your Model 100 to the home office computer, it is sent as a stream of individual ASCII characters, and it is recognized by the other computer one character at a time.

To communicate, computers must speak the same code, and they also must speak it in the same way. They have to be able to tell when a message begins and when it ends. They must know who is sending and who's receiving. They even have to know how fast the other computer is going to talk (transmit information), so that they can listen at the same rate of speed.

For these reasons, when the Model 100 communicates it has to be able to match the way that the computer on the other end of the line is communicating. When you use the TELCOM software you specify how the Model 100 will transmit data to another com-

puter—sort of like rules of grammar in English. We'll show you how later in this chapter.

These rules that computers follow when they communicate are called the communications protocols. Telecommunications has tons of jargon, even more than computer programming as a whole. There is an entire vocabulary of jargon that is used just to describe communications protocols. These words are listed below, with very brief explanations, and then described in detail in the next section. This is just supposed to give you the flavor of the jargon. Read further for the meaning of each parameter.

Parameter	General Meaning
Baud	Speed of transmission
Word length	Size of byte transmitted
Parity	A way of checking for errors
Stop bits	A way to keep track of characters
XON/XOFF status	A way of controlling transmissions
Dial pulse rate	Pulses per second when dialing

You don't have to know the technical meaning of each parameter to use TELCOM in most applications, such as CompuServe or Dow Jones. Just familiarize yourself with the parameter definitions. Then when they are used in the chapter you will understand what they mean, or you can refer back to them later. If you do want to know the technical meaning of each parameter, read the sections below. If you don't want to read the definitions now, skip to the heading *Connecting the Model 100 to Phone Lines*.

Baud

The baud is the rate of speed of data transmission between two computers. The baud is measured in *bits per seconds*. Most users of the Model 100 send and receive information through the modem: the built-in device that makes it possible to connect the Model 100

to the phone lines and transmit information. More about the modem later.

The Model 100's built-in modem uses 300 baud exclusively. That means that 300 bits of data are sent or received every second.

Word Length

Each ASCII character is called a word. Calling ASCII characters a word doesn't make sense if you think of one character as a letter of the alphabet and one word as a collection of these letters. But it does make sense when you know that each ASCII character is made up of 7 individual elements, called bits, and that in computer jargon this makes up a word. To a computer, the ASCII characaters for the upper case letters A through C look like this:

A - 01000001
B - 01000010
C - 01000011

The lines of 1s and 0s in the right column are the words for each character. Each 1 or 0 is a single bit. Notice that the left-most bit of the ASCII word is always a zero. ASCII characters only use seven bits of the word. The total word length used by many microcomputers is eight bits, with the left-most bit unused by the ASCII code. Therefore the ASCII word length is seven. The unused bit is available for parity checking, which is described below.

Parity Checking

Parity checking is used by computers to catch errors in data transmission. It's important to check for transmission errors because telephone lines are full of electrical "noise" that can create errors in the messages you send. Why check for these little errors, you might ask? Let's say you send this message from your Model 100 to your home office:

```
Customer agrees to pay $2 million.
```

The difference between the ASCII character for the numeral 1 and the ASCII character for the numeral 2 *is only one bit,* as shown here:

1 - 00110001
2 - 00110010

So if only one bit (a single 1 or 0 of the message) is transmitted in error, your message could read:

```
Customer agrees to pay $1 million.
```

Now that one bit really made a difference! That's why parity checking is used.

Computers can check for either even or odd parity. In *even* parity checking the sending computer places either a 1 or a 0 in the unused eighth bit of the eight-bit word in order to transmit an even number of 1s. The receiving computer keeps a running total of the number of 1s received. If the number of 1s is not even, an error message is displayed. A parity error only indicates that an error occurred during the transmission. It doesn't tell you when or where the error took place.

In *odd* parity checking, the total number of 1s is odd. When you consider that computers accomplish everything arithmetically, it is easy to see why parity checking is important. Since data is sent as a stream of bits (1s or 0s), one missing bit can throw off the entire sequence. If the sequence of bits is off, the data received is garbled.

Stop Bits

The stop bits are the number of bits that mark the end of a character. Because computers talk to each other in long sequences of 1s and 0s, they have to be able to tell when transmission of each ASCII character starts and stops. The beginning and end of transmission is signalled by the stop bits.

Line Status, XON/XOFF

XON (transmitter ON) and XOFF (transmitter OFF) are characters transmitted to control the flow of data between computers. Not all telecommunications software uses the XON/XOFF protocol.

Pulse Rate

The Model 100 will dial telephone numbers for you. The pulse rate determines the dialing speed and must be compatible with your telephone system. You can find out what the pulse rate used on your telephone is by calling your local telephone company.

Connecting the Model 100 to Phone Lines

To use TELCOM, connect the Model 100 to the telephone lines as shown in the owner's manual. Be sure to tell your telephone company the information listed in the owner's manual *before* connecting the Model 100 to the phone lines. This includes the manufacturer, Model, FCC ID number, FCC Registration number, and Ringer Equivalence Number.

You have two connection options, the modem cable and the acoustic coupler. These devices are plugged into the PHONE connector in the rear of the Model 100 case and then connected to the telephone equipment.

The modem cable, shown in Figure 80, attaches the Model 100 directly to the phone lines using universal telephone plugs. But a universal telepone plug is not always available. If you are sending data from a public telephone, you can't just unplug the phone from the phone booth and plug in your Model 100. The telephones in hotel rooms, where the portable Model 100 is likely to be used, are often wired directly to the wall outlet.

To overcome these problems you can use the acoustic coupler, shown in Figure 81. The acoustic coupler looks like a pair of stereo headphones for the telephone. A pair of lightweight foam cups fits over the ear and mouth pieces of the telephone handset. With the

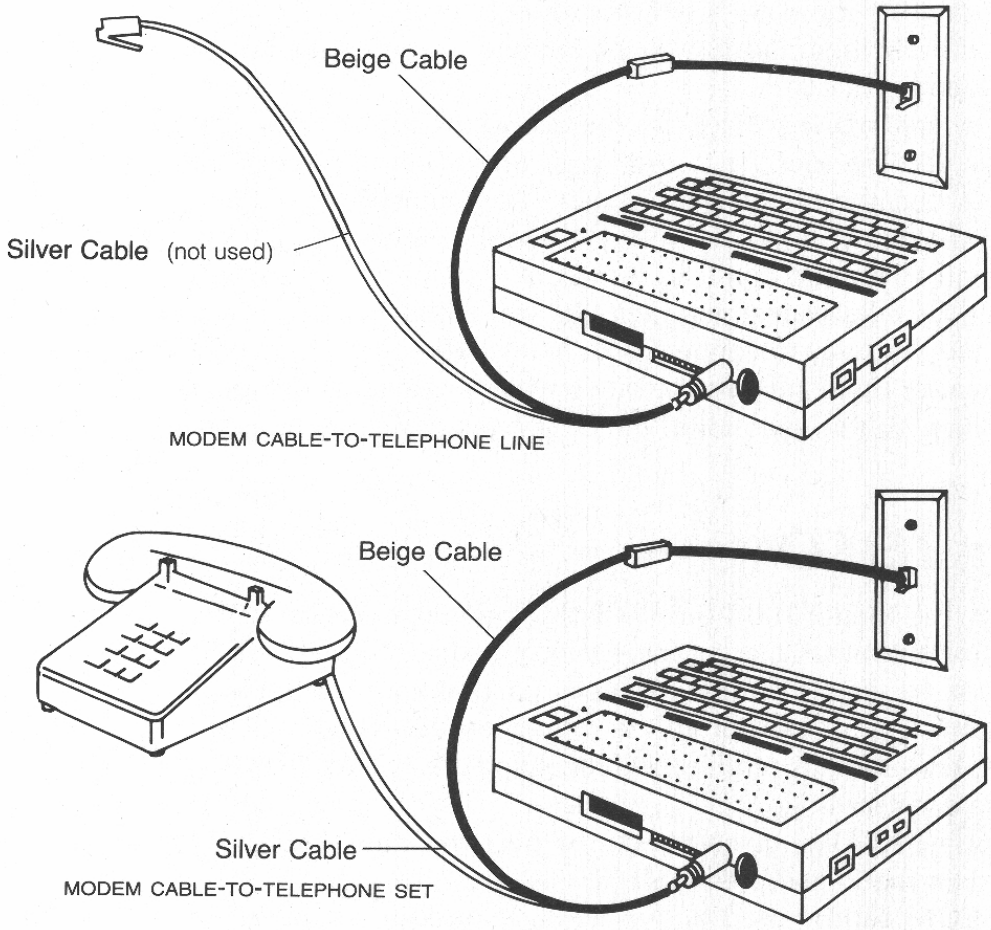


Figure 80 MODEM Cable-to-Telephone Line and MODEM Cable-to-Telephone Set

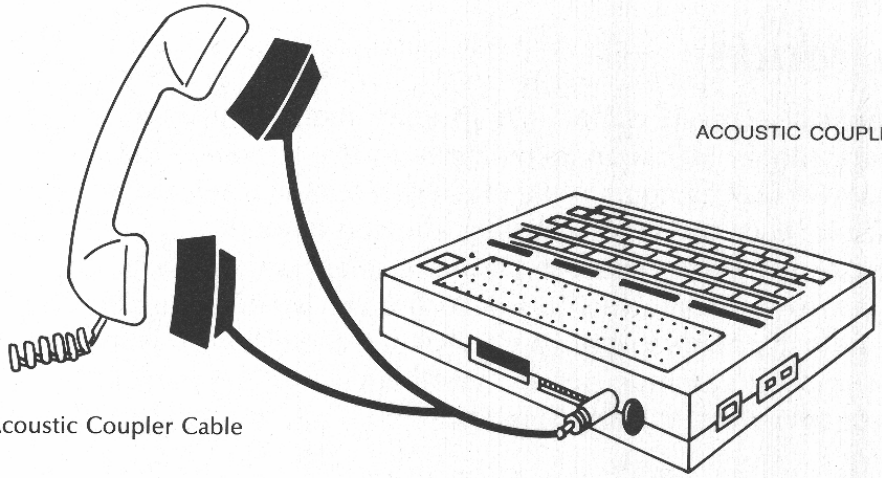


Figure 81 Acoustic Coupler Cable

acoustic coupler the data is transmitted as a series of beeps and boops, actually high and low tones audible to the computer on the other end of the line.

There are two drawbacks to the acoustic coupler: First, the computer you are sending to may miss one of the thousands of tiny beeps because of background noise not muffled by the foam cups; and second, many of today's telephone handsets have streamlined styles that don't accommodate the foam cups. Try to anticipate the telephone equipment that you will be using and determine the best way to connect your Model 100 before you take it on the road. There are only these two ways to connect the Model 100 modem to the telephone: direct connect cable or acoustic coupler.

Using TELCOM

TELCOM, the Model 100's built-in telecommunications software, is a computer program that manages telecommunications. TELCOM allows you to set the communications parameters we discussed earlier, and it takes care of all the nitty-gritty details of message transmission—such as sending and receiving the ASCII codes that comprise each message.

To use TELCOM, place the cursor over the name TELCOM on the main menu and press ENTER.

TELCOM can be used in two ways, referred to as modes of operation. They are called Entry mode and Terminal mode.

Entry Mode

When you first enter TELCOM it is in Entry mode. Entry mode converts the Model 100 to an autodialer, just like a memory telephone that will dial numbers at the touch of a button. The modem cable must be used for the autodialing function to work.

You can use entry mode to dial your telephone when you want to talk to someone, as you would use any autodialer. You can also use entry mode to dial your telephone and establish computer-to-computer communications. We'll have a lot more to say about these two ways of using TELCOM.

Terminal Mode

When TELCOM is used in Terminal mode the Model 100 can communicate with other computers and take advantage of all the benefits of telecommunications mentioned earlier in this chapter. In Terminal mode the Model 100 can send and receive ASCII files, display or print information from another computer, and send and receive electronic mail.

It is also possible to go directly from the autodialing feature of Entry mode to Terminal mode by using the automatic log-on sequence described later. This lets you dial the number of another computer or data base and begin to communicate.

TELCOM, in Terminal mode, makes the Model 100 function as a terminal of *another computer*. This allows you to use the facilities, such as the data files, of the other computer. When the Model 100 is in terminal mode the other computer is called the host.

So let's look at what TELCOM can do.

Dialing with TELCOM Entry Mode

When you place the main menu cursor over the name TELCOM and press ENTER, the Model 100 looks for the ADRS.DO file that contains the names and telephone numbers of the people or computers you wish to call. Therefore, to use Entry mode you must have created the ADRS.DO file as shown in the owner's manual. The telephone numbers in the ADRS.DO file must begin and end with a colon (:). A sample ADRS.DO file is shown in Figure 82.

```
Mr. Smith,A.B.C. Inc.,:2125551212:
Ms. J. Jones,Electro Corp.,:2125551221:
Mr. N. Gineer, Computer Co.,:2125551234:
Ms. A. Name,X.Y.Z. Inc.,:2125554321:
```

Figure 82 Sample ADRS.DO File

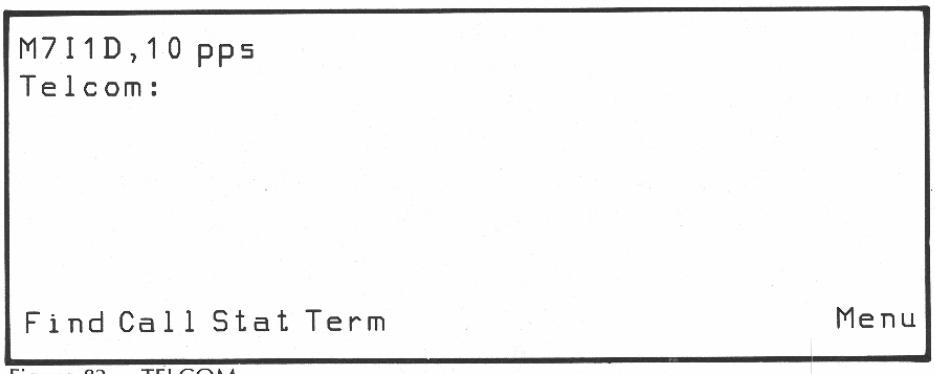


Figure 83 TELCOM

When you first start to use Entry mode the display screen looks like Figure 83. To manually dial a telephone number press key F2, Call.

TELCOM displays a blinking cursor, and you type in the number you want to call and press ENTER. The Model 100 dials the number for you. The message CALLING appears on the next line of the display, along with each digit of the number being called as it is sent over the phone lines by the Model 100:

```
Calling 5551234
```

If the Model 100 modem cable is attached to the back of a telephone, you must lift the telephone handset while TELCOM is dialing. Figure 84 shows the screen as it appears after manual dialing.

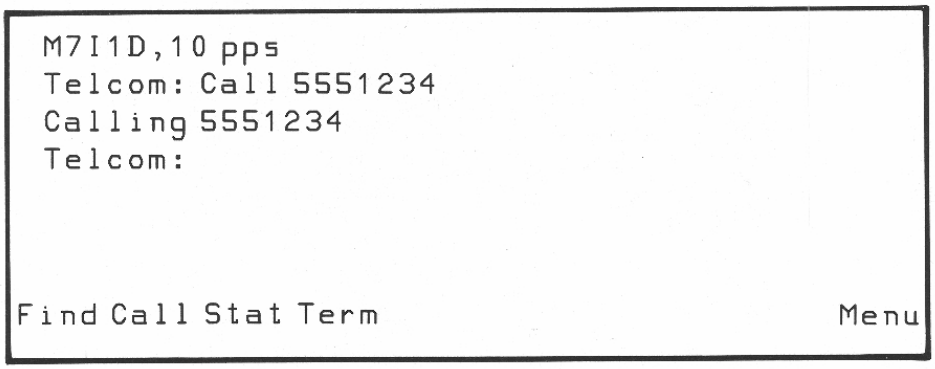


Figure 84 Manual Dialing

To use TELCOM to automatically dial numbers that you have stored in the ADRS.DO file, first press key F1, Find. Then type the string of letters or numbers that will help TELCOM find the number in the ADRS.DO file. For example, if you want to find the telephone number of Mr. Smith, at A.B.C. Inc., you could enter SMITH after the Find prompt.

```
Telcom: Find SMITH (ENTER)
```

Find doesn't distinguish between upper and lower case, so you can use all uppercase if you like.

```
M7I1D,10 pps
Telcom: Find SMITH
Mr. Smith,A.B.C. Inc.,:2125551212




     Call More Quit
```

Figure 85 Autodialing, String Has Been Found

When the string you entered (SMITH) is found, the complete line from ADRS.DO file is displayed on the screen as shown in Figure 85.

```
Mr. Smith, A.B.C. Inc., :2125551212
```

TELCOM displays three options on the bottom line of the screen: Call, More and Quit.

- If the string you entered (SMITH) appears more than once in the ADRS.DO file you can press key F3, More, to bring the next instance of that name to the screen. You can decide again whether to Quit or Call.

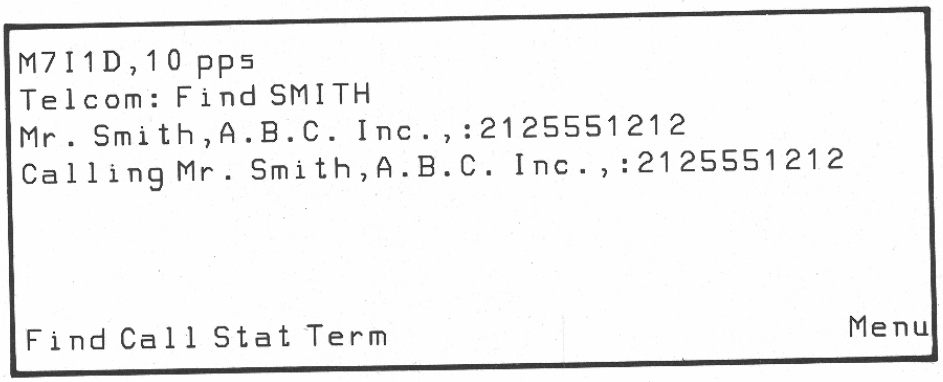


Figure 86 Dialing Complete

- If the number you want to dial has been found, simply press key F2, Call. TELCOM dials the number for you. Figure 86 shows how this appears on the Model 100 screen.
- If you don't want to dial the number displayed press key F4, Quit.

Whenever you use TELCOM's dialing features the Model 100 must be connected to modular phone lines with the modem cable. These features don't work with the acoustic coupler.

TELCOM Terminal Mode

When you want to use the Model 100 to communicate with another computer, information service, data bank, or bulletin board, use TELCOM's Terminal mode. Before entering Terminal mode you follow these steps:

1. Set the answer/originate switch on the left side of the Model 100 case. If you are originating the call to another computer, set the switch to ORIG. If you are receiving a call from another computer, set the switch to ANS.
2. Set the modem/Acoustic coupler switch on the left side of the case. Select DIR (Direct) if you are using the modem cable, ACP (Acoustic Coupler) if you are using the acoustic

coupler. This will tell the Model 100 whether it must generate the audible beeps and boops for the acoustic coupler.

3. Set the communications parameters, described earlier, to match those of the computer at the other end of the phone lines. The communications parameter settings to use to call most information services are listed in the second column below. These are fairly standard for major information services, but to be sure check first with your information service:

Parameter	Setting	Indicates
Baud rate	M	300 baud
Word length	7	7 bits
Parity	I	Ignore parity
Stop bits	1	1 stop bit
XON/XOFF Status	E	Enable
Dial pulse rate	10	10 pulses per second

You may want to communicate with a computer, or even an information service or bulletin board, that does not use the parameters shown above. Before you try to contact the other computer, call the person who will be operating the other computer and agree on the communications parameters that you will *both* use. This should be done each time because the operator of the other microcomputer may have changed the parameters. The full range of parameters that you can choose are shown in the Table in Figure 87.

Remember that the Model 100 MODEM only sends and receives at 300 baud. This limits the other party to 300 baud. Before you send important data, verify that you are using the correct parameters by sending and receiving a test file. A test file could be any file that you create with the TEXT word processor. If the test file can be transmitted correctly, you probably can assume that important data will make the trip unharmed.

	You Type:	For:
Baud	M	"modem" (300)
	1	75 baud
	2	110 baud
	3	300 baud
	4	600 baud
	5	1200 baud
	6	2400 baud
	7	4800 baud
	8	9600 baud
	9	19200 baud
Word Length	6	6 bits
	7	7 bits
	8	8 bits
Parity	I	Ignore parity
	O	Odd parity
	E	Even parity
	N	No parity
Stop Bit	1	1 stop bit
	2	2 stop bits
Line Status	E	Enable (XON)
	D	Disable (XOFF)
Pulse Rate	10	10pps
	20	20pps

Figure 87 Model 100 Telecommunications Protocol

Changing Communications Parameters

TELCOM communications parameters can be changed while you are in Entry mode. After selecting Entry mode, press key F3 (Stat) and type in the new parameters, with no spaces between each parameter, in this order: baud, word length, parity, stop bits, XON/XOFF status, dial pulse rate. The display screen created by this procedure is shown in Figure 88.

```
M7I1E,10 pps
Telcom: Stat M802D
Telcom: Stat
M802D,10 pps
Telcom:


Find Call Stat Term                    Menu
```

Figure 88 Changing Communications Parameters

Entering Terminal Mode

To enter the Terminal mode from Entry mode you have to contact the host computer by telephone and then press function key F4 (Term). This can be done manually or automatically, as described below.

Entering Terminal Mode Manually

```
Prev Down  Up  Full Echo                Bye
```

Figure 89 Terminal Mode Function Key Labels

When modular phone lines are not available, or when you are using the Acoustic Coupler cable, follow the manual method. Set the answer/originate switch to ORIG. Using the Entry mode of TELCOM, lift the telephone handset and dial the host computer's telephone number. When the host computer answers, the Model 100

will emit a high-pitched tone. Press key F4 (Term). If you are using the Acoustic Coupler, place the foam cups over the ear and mouth pieces. If you are using the modem hang up the handset. You know you've entered Terminal mode when the label line displays the Terminal mode definitions for the functions keys, shown in Figure 89.

Entering Terminal Mode Automatically

You can enter Terminal mode automatically if you are using the modem cable. To do so, store the host computer's telephone number in the ADRS.DO file followed by the "less than" and "greater than" symbols. Terminal mode uses these symbols when dialing the number.

For example, if the CompuServe telephone access number were 555-1234, you could store it in ADRS.DO as:

```
COMPUSERVE :5551234<>:
```

To enter Terminal mode automatically, set the answer/originate switch to ORIG. In TELCOM's Entry mode use the Find feature, key F1, to locate the telephone number you wish to call. When you have the number on the screen, press key F2 (Call).

TELCOM dials the number. In most cases you will hear the phone ringing, or a busy signal, as TELCOM tries to contact the host. After the host is contacted, the Model 100 emits a high-pitched tone. The Model 100 lets you know that the number has been dialed and that it is now in Terminal mode by displaying the function keys Terminal mode definitions on the label line (see Figure 89).

Auto Log-On Sequence

A special feature of automatic entry into Terminal mode is automatic log on. When you contact information services, such as CompuServe, you are asked a series of questions that you must answer in order to verify your user-identification number and password. Typing in this information every time you call the information service would be tedious. Fortunately TELCOM lets you create an

Auto Log-On Sequence that answers the log-on questions for you.

The Auto Log-On Sequence directs the Model 100 to give your ID and password to the host. The Auto Log-On Sequence is stored after the telephone number of the host in ADRS.DO. It is placed between the "less than" and "greater than" symbols.

For example, if the CompuServe telephone access number were 555-1234, you could store it in ADRS.DO, followed by the Auto Log-On Sequence, as:

DOW JONES:1555123242=<C?U[ID#]^?P[password]^3M>4:

Complete Auto Log-On Sequences for CompuServe and Dow Jones Information Service are provided with the Model 100 modem cable literature.

Prev

When in Terminal mode the information transmitted by the host will appear on the screen of your Model 100 as it is sent. If you want to review lines of text that have scrolled up out of sight, you can press key F1, (Prev), to bring the previous eight lines of information back into view.

Echo

As information is displayed on the Model 100 screen by the host, it can be printed on a printer. Simply press key F5, Echo, while in Terminal mode.

Many of the data services provide much more information than anyone can use. As this stream of text scrolls up the Model 100 display, the Echo function can also be used to make a quick hard copy of anything that catches your eye. Pressing the Echo key once turns the printer on. Pressing it again turns the printer off.

You can use the Echo features at any time that you are receiving a transmission from the other computer. One advantage to the Echo function is that large amounts of text can be saved for review without occupying any of the RAM memory of the Model 100.

This can be useful in a number of ways. For example, the larger data services, such as CompuServe, will send you documentation that includes user instructions and copies of the menus.

But local bulletin boards usually only supply this information on-line. You can create your own user manual by making a quick hard copy of on-line information with Echo.

Full Duplex and Half Duplex

Function key F4 lets you toggle between Full Duplex and Half Duplex communications. These terms are part of the opaque jargon that abounds in the world of computers, especially in telecommunications. They merely describe the way the host and the Model 100 transmit data back and forth. As with communications parameters, you must set TELCOM to match the communications method used by your host computer, either Full Duplex or Half Duplex.

Transferring Files Between Computers

In Chapter 5, we said that files can be transferred between the Model 100 and other computers. With TELCOM you can transmit text files created with the TEXT word processor and receive text files, including letters, memos, technical specifications, proposals, and more. You can also send and receive data files, such as the MAIN.DO and the PRICE.DO data files used in the Order Entry program, Figure 62, Chapter 5. You can send data about orders, merchandise inventories, parts inventories, or repair call reports. You can receive current price, order, and customer information at any time.

Electronic mail is the computer's equivalent of a post office box. Both information services and bulletin boards allow you to leave messages for others and to pick up messages left for you. To open your electronic mail box, you type in your name or password, then read your mail on the Model 100's screen or use the Echo feature to have the mail typed out on a printer.

If you are on the road, your secretary can leave electronic mail for you on CompuServe; you can access this information at any time of the day or night and save the mail in a data file. The mail that you wish to send can be composed with TEXT and simply transmitted to the data bank as a file rather than being typed in

line by line. This saves time because you only type the document once. It saves money, because only time spent on the information service line is charged to your account, not the time you spent typing. It also lets you send documents error free because they can be proofread on the display screen before they are transmitted.

This sending and retrieving of files is called downloading and uploading. To download means to store information transmitted to your Model 100 by a host computer. To upload means to transmit an existing file from the Model 100 to a host computer. The download and upload features of TELCOM are initiated with the function keys. The TELCOM software handles the details of sending or receiving the data and creating the file. These are described below.

Download

To download incoming information, press key F2 (Down). The Model 100 will display this prompt:

```
File to Download?
```

You then enter the name you'd like to use for the file that will store the incoming information. If you are receiving a sales memo you could name the file SMEMO. TELCOM supplies the .DO extension if you omit it:

```
File to Download? SMEMO (ENTER)
```

The data will be stored in the file, SMEMO.DO, as it is received. When the entire document has been received, press key F2 again to stop downloading. Be warned—if the Model 100 does not have enough RAM to store the entire transmission, downloading stops automatically.

Once the incoming document is downloaded it can be used like any other text file. It can be printed out, merged with other text files, or displayed on the Model 100 screen.

And when data files have been downloaded they can be used by BASIC programs. An up-to-date PRICE.DO file, such as the file shown in Figure 6.6, could be transmitted to you and then used by the Order Entry program in Figure 62. If you travel, new

ADRS.DO files could be transmitted to you as you move from one area to another. The only limit to the use of transmitted data files is your imagination and the amount of RAM in the Model 100.

```
File to Upload? SALES.DO (ENTER)
```

Upload

To upload a file press key F3, Upload. TELCOM displays this prompt on the screen:

```
File to Upload?
```

You then type in the filename of the file you want to transmit. To upload the file, SALES.DO, a sales report, type:

File to Upload? SALES.DO (ENTER)

TELCOM will then respond with a prompt that asks you the width of the lines you want to send:

```
Width:
```

If you are sending a data file, or another file that you wish to send unchanged, *only press ENTER*. If you only press ENTER the file is sent *exactly* as exists in memory:

```
Width: (ENTER)
```

If you are sending a document file, you can respond with the number of characters per line of the host computer screen. The Model 100 has 40 characters per line. If the computer you are sending to has an 80-column display you may want to respond with **80:**

```
Width: 80 (ENTER)
```

Then press ENTER. TELCOM will transmit the file, SALES.DO, with 80-character lines to the computer on the other end of the phone line.

As with Download, data files can also be transmitted with Upload. The MAIN.DO file, from Figure 67, could be sent to the home office, where the order confirmation could be printed and sent to the customer.

Bye

To end communication with a host computer press key F8, Bye. The Model 100 will ask if you want to:

```
Disconnect?
```

Press 'Y', and then press ENTER.

```
Disconnect? Y (ENTER)
```

You will return to TELCOM's Entry mode. Then press key F8, Menu, to return to the Model 100 main menu.

It's important to enter and exit TELCOM properly. When you are using TELCOM the Model 100 has control of the telephone lines. If you do not use TELCOM correctly you may disrupt your telephone service.

If you don't connect the telephone cable correctly and enter TELCOM correctly you can also get the Model 100 "hung up" in TELCOM waiting for a signal from the other computer.

You can also disrupt the activities of the host computer if you don't begin and end communications properly. Be sure to see your information service user's guide for the proper log-on procedures.

Chapter 7

Programming Graphics and Sound

Although BASIC programs for business let you manipulate and store data, what you want is the end result—the output.

And when you plan to show the output to someone important, like your boss or a customer, you want it to look great. With Model 100 BASIC you can create eye-catching displays and print-outs that will inform and influence others. Model 100 BASIC has special commands, like **PRINT** @ and **PSET** that give you complete control over every point on the display screen. And with built-in graphics characters you can create just about any display that you can imagine.

When you use the Model 100's built-in sound generator, even simple messages become attention-grabbers. The sound generator produces tones over a full five octaves. You can program it to play familiar tunes or to sound like an army of invading Martians.

We've already covered many BASIC commands that let you format output. These include CLS, PRINT USING, TAB, and the use of semicolons and commas after PRINT statements. But there's more. This chapter presents the remaining commands and winds up with a sample presentation program. This program turns your Model 100 into a briefcase-sized portable sales pitch.

Print @

When you use PRINT statements, BASIC automatically determines where to display the text on the screen. Starting with a clear screen, BASIC PRINTs one line at a time and scrolls up when the text reaches the bottom of the screen. The **PRINT** @ statement lets you display something in the middle of the screen, even if there is already information shown elsewhere on it. With the **PRINT** @ statement you can direct the Model 100 to display characters in any of the 320 character positions on the screen. It follows the form,

PRINT @ *screen position, "text string"*

where the screen position is an integer from 0 to 319. The comma

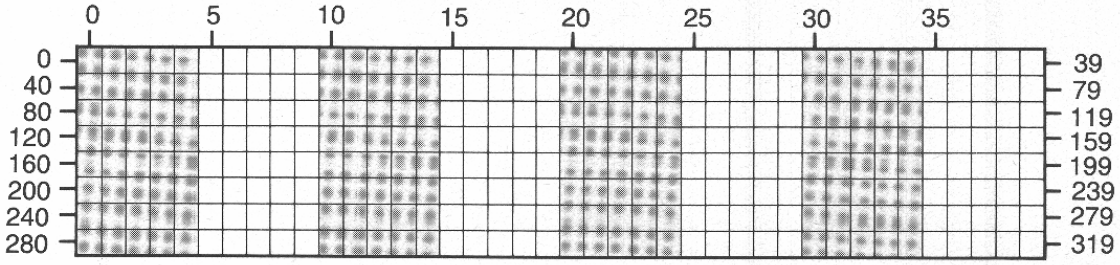


Figure 90 Improved Screen Layout Form

after the screen position is required for correct syntax. The text string in quotes works just as the PRINT statement does.

The Improved Screen Layout Form, in Figure 90, shows the 320 (0-319) screen positions on the Model 100 display. Each position can hold one character: a letter, a number, or a symbol.

To get the number for any position, add the number on the top to the number on the left. For example, the screen position for the eleventh column (10) of the fourth row (120) is 130 (10+120). To display the word "DISPLAY" starting at position 130 use the statement:

```
PRINT @ 130, "DISPLAY"
```

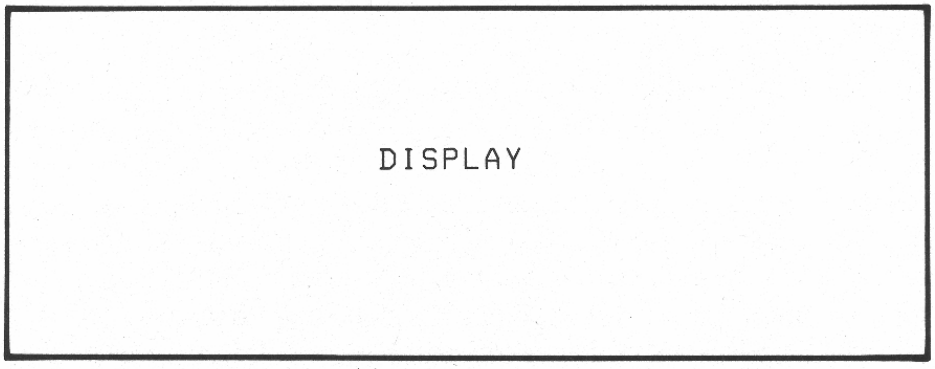


Figure 91 Using the PRINT@ Statement

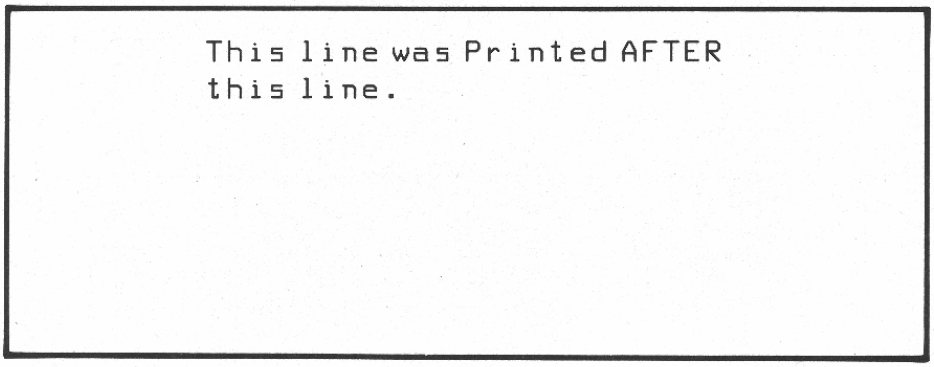


Figure 92 Using the PRINT@ Statement

This is shown in Figure 91. Figure 92 shows what happens when the following program is executed.

```
10 CLS
20 PRINT @ 130, "this line."
30 PRINT @ 90, "This line was printed
AFTER"
```

As you can see, **PRINT @** lets you put things anywhere on the screen in any order. It could be used for displaying the date, the time, or some other value in one constant screen position while other information is displayed on the screen. It also can be used to create graphic effects, such as titles displayed in the middle of a textured background.

PSET and PRESET

The **PSET** and **PRESET** commands of Model 100 BASIC give you even more control over the display screen than you get with PRINT @. The Model 100 screen is divided into 15,360 individual pixels. Pixel is an acronym which stands for PICture ELement.

If you look at the display screen closely, you can see that each character is composed of many of these tiny black pixel squares. These pixels perform the same function as the dots that make up a TV image or a newspaper photograph. The pixels on the Model 100 LCD screen are even more evident than they are on most CRT computer display terminals. Each pixel can be *set,* turned on, or *reset,* turned off. When a pixel is turned on, it is visible.

The pixels are located with their X-Y coordinates. The X coordinate indicates the horizontal position. There are 240 possible horizontal positions from 0 to 239. The Y coordinate indicates the vertical position. There are 64 possible vertical positions from 0 to 63. The Graphics Worksheet, in the Model 100 user's manual, shows how the pixels are numbered horizontally and vertically.

PSET

The **PSET (P**ixel **SET)** command turns on any pixel. It takes the form:

PSET *(X coord,Y coord)*

where *X coord* and *Y coord* define the pixel that will be turned on. The command:

PSET (0,0)

turns on the one pixel in the upper left hand corner of the screen. The command:

PSET (120,32)

turns on one pixel in the center of the screen. The short program that follows turns on every other pixel in the display, about 7,680 pixels, starting in the upper left-hand corner. The result is a polka

dot pattern. Turning on individual pixels takes *a lot* of time. The program below takes 25 seconds:

```
10 CLS
20 FOR Y=0 TO 63 STEP 2
30 FOR X=0 TO 239 STEP 2
40 PSET (X,Y)
50 NEXT X
60 NEXT Y
70 END
```

PRESET

The **PRESET (P**ixel **RESET)** command works just like the PSET command, but it turns the pixel off, making it invisible. It follows the same format:

PRESET *(X coord, Y coord)*

where *X coord* and *Y coord* are the location of the pixel that will be turned off. You probably won't be turning on and off individual pixels, unless you feel like really challenging yourself. But BASIC also gives you the LINE command to draw lines and boxes.

LINE

The **LINE** command lets you draw lines and boxes made of pixels. The boxes created can be white or black, solid or outline.

To draw *lines* use the form:

LINE (X1, Y1)-(X2,Y2),*color switch*

The values of (X1,Y1)-(X2,Y2) are the two ends of the line. The *color switch* is used only in the **LINE** command. If the color switch is 0, the pixels of the line will be turned off. If the color switch is 1, the pixels of the line will be turned on. If no color switch is specified the pixels will be turned on.

The command:

```
LINE (0,0)-(239,63)
```

draws a line from the upper left corner of the screen, (0,0), to the bottom right corner (239,63).

To draw *boxes* use the form:

LINE (X1,Y1)-(X2,Y2),*color switch*,**BF**

If the **B** is added to the end of the command a box is drawn whose opposite corners are located at the points defined by (X1,Y1)-(X2,Y2). If the **F** is also added to the command, the box is filled in with the color indicated by the color switch. The color switch works in the same way: 0 turns the pixels off, 1 turns them on. When drawing boxes you must specify the color switch.

The command:

```
LINE (0,0)-(239,63),1,B
```

draws an outline box whose corners are the four corners of the screen. And the command:

```
LINE (0,0)-(239,63),1,BF
```

draws a solid box in the same position, making the entire screen black. These commands are used in a program at the end of this chapter, and there are photographs that show the results.

To draw a line between the upper left and lower right corners of the screen, use the command:

```
LINE (6,8)-(233,55),1,B
```

Graphics Characters

The Model 100 graphics characters are listed in the table of ASCII codes in Appendix A. The graphics characters have ASCII numbers from 128 to 176 and from 225 to 255. The characters from 177 to 223 are mostly foreign language alphabet characters.

The characters from 128 to 176 are symbols, such as an airplane, a telephone, a space ship, and a little man. There are also traditional graphics characters such as the suits of a deck of cards, the male and female symbols, and mathematical symbols. The

graphics characters from 225 to 255 can be used to construct borders, boxes, and outlines.

The Model 100 graphics characters are *not* really ASCII characters. The ASCII characters 0-127 represent the letters, numbers, symbols, and control characters (carriage return is a control character) that are used to display or print text. The graphics characters in the ASCII character chart are unique to the Model 100. They are called the Model 100 *extended* ASCII character set.

Three Ways to Use Graphics

With BASIC you can display graphics characters three ways. You can display the graphics characters directly by simultaneously pressing the **GRPH** key and the alphanumeric key for the graphic character that you want. For example, the keys to press for the telephone symbol are **GRPH p.** The keys to press for any graphic symbol are listed in the right-hand column of the ASCII codes chart.

You can also use graphics characters in PRINT statements. The command:

PRINT *"type GRPH p between quotes"*

would send the symbol for the telephone to the screen. To produce the telephone symbol in the PRINT statement, type *GRPH p* between the quotation marks.

You can also display the telephone graphic character on the screen with the command:

PRINT CHR$(128)

because the decimal value for the telephone symbol in the ASCII code chart is 128.

If you would like to see all of the Model 100 graphics characters displayed on the screen, type in this little program and run it:

```
10 CLS
20 FOR X=128 to 176
```

```
30 PRINT CHR$(X);" ";
40 NEXT X
50  PRINT:PRINT
60    FOR X= 225 TO 255
70    PRINT CHR$(X);" ";
80    NEXT X
```

This program takes advantage of the fact that you can display a graphic character on the screen with the command:

PRINT CHR$ *(ASCII number for graphic character)*

The FOR . . . NEXT loop in lines 20–40 displays the graphics characters from ASCII 128 to ASCII 176. The second FOR . . . NEXT loop, lines 60–80 displays the graphics characters from ASCII 225 to ASCII 255.

SOUND

The sound generator is controlled by the **SOUND** command:

SOUND *pitch, length*

	Octave				
Note	**1**	**2**	**3**	**4**	**5**
G	12538	6269	3134	1567	783
G#	11836	5918	2959	1479	739
A	11172	5586	2793	1396	698
A#	10544	5272	2636	1318	659
B	9952	4976	2488	1244	622
C	9394	4697	2348	1174	587
C#	8866	4433	2216	1108	554
D	8368	4184	2092	1046	523
D#	7900	3950	1975	987	493
E	7456	3728	1864	932	466
F	7032	3516	1758	879	439
F#	6642	3321	1660	830	415

Figure 93 SOUND Pitch Table

The *pitch* is a number from 0 to 12538. The lowest number represents the highest *pitch*. The *length* is a number from 0 to 255. You can estimate the duration of the tone in seconds by dividing *length* by 50. *Length* 150 would be about 3 seconds. The maximum *length* of 255 would be about 5 seconds.

The tones produced by the sound generator are shown in the table in Figure 93. The command:

SOUND 4697; 150

will produce the note middle C for about three seconds. The program below plays the familiar tune, "Mary Had a Little Lamb."

```
10 SOUND 4697, 20
20 SOUND 4976, 20
30 SOUND 5586, 20
40 SOUND 4976, 20
50 SOUND 4697, 20: SOUND 4697, 20: SOUND
4697, 40:
```

Presentation Program

The Presentation Program, Figure 94, turns the Model 100 into a portable sales pitch. The subject of the sample Presentation Program is this book. But you can modify the program to make a presentation about any subject you wish. Once you write and debug your own presentation program you are ready at any time to set the Model 100 in front of the person you wish to impress and start the show. Like any good salesperson, the Presentation Program emphasizes the features and benefits of the product.

```
10 'PRESENTATION PROGRAM CREATES A
20 'PORTABLE SALES PITCH
30 '---------------FIRST FRAME
40 CLS
50 LINE (0,0)-(239,63),1,BF
60 PRINT @ 127,"GETTING WHAT YOU WANT FROM"
70 PRINT @ 170,"THE TRS-80 MODEL 100"
80 GOSUB 1000
90 GOSUB 2000
```


Figure 94 continues

```
100 '---------------SECOND FRAME
110 LINE (30,24)-(209,39),0,BF
120 PRINT @ 125,"BASIC PROGRAMMING FOR
BUSINESS"
130 PRINT @ 173,"BY E. PAUL CONE"
140 LINE (24,16)-(215,47),0,B
150 LINE (12,8)-(227,55),0,B
160 GOSUB 2000
170 GOSUB 1000
180 '---------------THIRD FRAME
190 FOR X=0 TO 318
200 PRINT @ X,CHR$(147);
210 NEXT X
220 LINE (30,24)-(209,39),0,BF
230 PRINT @ 125, "EVERYONE IS READING IT"
240 PRINT @ 185, "BECAUSE..."
250 GOSUB 2000
260 CLS '----------FOURTH FRAME
270 PRINT @ 43,"THIS BOOK HELPS MODEL 100
USERS TO:"
280 PRINT:PRINT @ 83,"1-LEARN BASIC
PROGRAMMING."
290 PRINT @ 123,"2 - SOLVE BUSINESS
PROBLEMS."
300 PRINT @ 163,"3 - STORE AND RETRIEVE
INFORMATION."
310 PRINT @ 203,"4 - USE M-100
TELECOMMUNICATIONS."
320 PRINT @ 243,"5 - STAY AHEAD OF THE
COMPETITION!"
330  FOR X=1 TO 38
340  PRINT @ X,CHR$(239);
350  NEXT X
360   FOR X=79 TO 279 STEP 40
370   PRINT @ X,CHR$(239);
380   NEXT X
390   FOR X=318 TO 281 STEP -1
400   PRINT @ X,CHR$(239);
```

Figure 94 continues

```
410    NEXT X
420    FOR X=240 TO 40 STEP -40
430    PRINT @ X,CHR$(239);
440    NEXT X
450 GOSUB 2000: GOSUB 2000
460 '---------------FIFTH FRAME
470  FOR X=0 TO 318
480  PRINT @ X,CHR$(239);
490  NEXT X
500 PRINT @ 122,"THE BOOK"
510 PRINT @ 162,"EXPLAINS"
520 LINE (60,24)-(90,16),0
530 BEEP: PRINT @ 55, "BASIC COMMANDS   "
540 LINE (60,32)-(90,32),0
550 BEEP: PRINT @ 135,"BUILT-IN FEATURES"
560 LINE (60, 40)-(90,48),0
570 BEEP: PRINT @ 215,"UNIQUE FUNCTIONS"
580 GOSUB 2000
590 '---------------SIXTH FRAME
600  FOR X=0 TO 318
610  PRINT @ X,CHR$(255);
620  NEXT X
630 LINE (30,24)-(209,39),0,BF
640 PRINT @ 135," READ IT"
650 PRINT @ 177,"TODAY!"
660 LINE (24,16)-(215,47),1,B
670 LINE (12,8)-(227,55),1,B
680 GOSUB 1000:GOSUB 2000
690 GOTO 30
1000 '---------------SOUND SUBROUTINE
1010 SOUND 4697,10
1020 SOUND 2348,20
1030 SOUND 1174,30
1040 RETURN
2000 '---------------DELAY SUBROUTINE
2010 FOR DL = 1 TO 2000
2020 NEXT DL: RETURN
```

Figure 94 Presentation Program

The photographs of the screens don't really convey the effect of the program. It is actually animated, combining the movement of lines, boxes, and text with sound. Type it into the Model 100 and see for yourself.

The Presentation Program uses the LINE and PRINT @ statements to create six frames, as if it were presenting an animated slide show. As you read the following description of the program you will notice that a small number of commands are used as modules to create a number of effects.

First Frame

Line 50 sets all the pixels to turn the screen black. Then Lines 60–70 use PRINT @ statements to position the title of the book in the center of the screen, as shown in Figure 95.

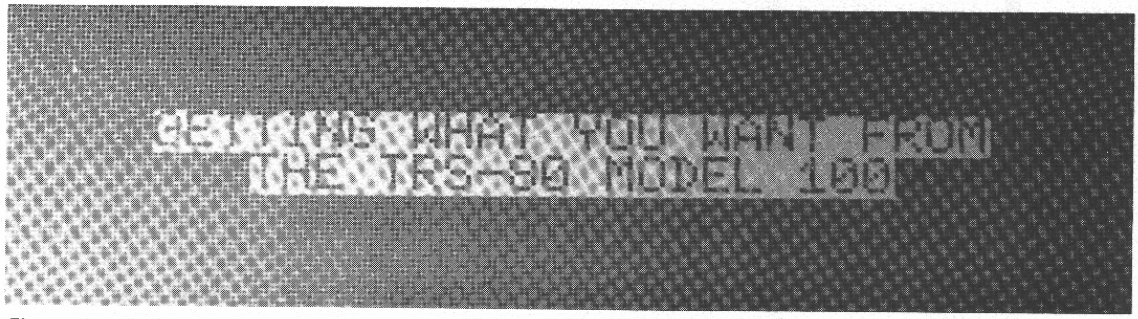


Figure 95 The Announcement Banner for a Presentation Program

Line 80 calls the subroutine in lines 1000–1040. This sound subroutine produces a series of three attention-getting tones. Line 90 calls the subroutine in lines 2000–2020. This delay subroutine introduces a new use for the FOR . . . NEXT loop. A delay routine is a FOR . . . NEXT loop that does nothing. There are no lines of code between the FOR and the NEXT commands. It is used to slow down, or delay, execution of the program to give the user time to do something, such as read the information on the screen. The parameters of a delay loop can be set by trial and error. If the loop does not produce a long enough delay, just increase the ending loop parameter. On the Model 100 a delay loop that executes 350 times lasts about one second.

The loop in lines 2010–2020 is executed 2,000 times, long

enough for an average reader to read the titles. On frames with more text, the delay subroutine is called two times in a row to produce a longer delay.

Second Frame

Line 110 uses the LINE command to draw an empty box, erasing the first title. This leaves the background intact. Then lines 120–130 place the subtitle and the author's name neatly in the box. Lines 140–150 draw two concentric outline boxes to add a graphic quality to the frame, as shown in Figure 96.

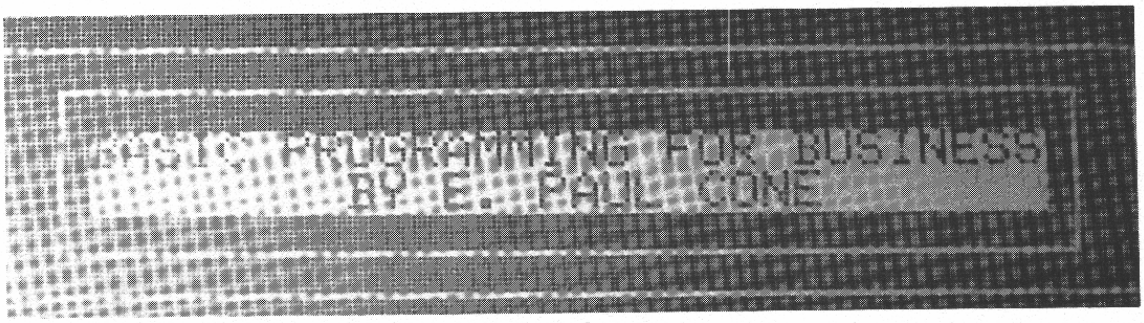


Figure 96 The Second Frame of the Presentation Program

Third Frame

The third frame uses the graphics character 147 in the Model 100 ASCII codes chart—the little man. The FOR . . . NEXT loop in lines 190–210 starts at the first screen position, 0, and uses the PRINT @ statement until the little man fills the screen.

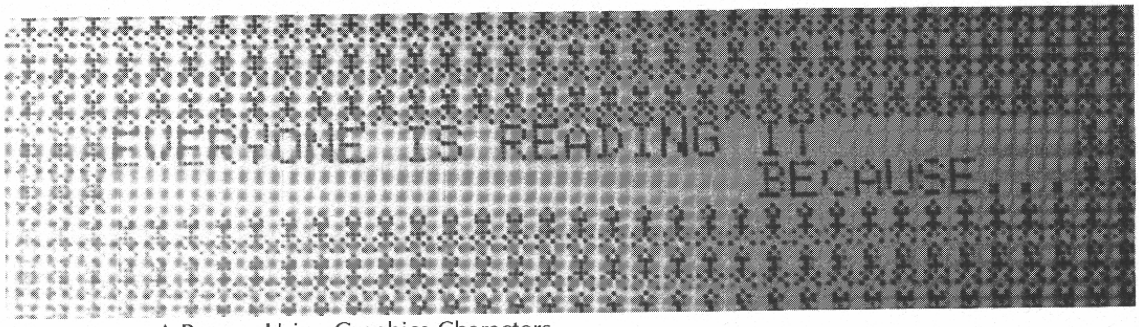


Figure 97 A Banner Using Graphics Characters

The semicolon at the end of the PRINT @ statement is important. If it's not there the carriage return interferes with the placement of the little man on the screen.

Lines 220–240 create the empty box and place the new title, "EVERYONE IS READING IT BECAUSE . . .," inside it, as shown in Figure 97.

Fourth Frame

The fourth frame lists the book's main features with lines 270–320. This frame is mostly text. A border is created by the four FOR . . . NEXT loops in lines 330–440. These loops use the same principle as the little man loop in lines 190–210.

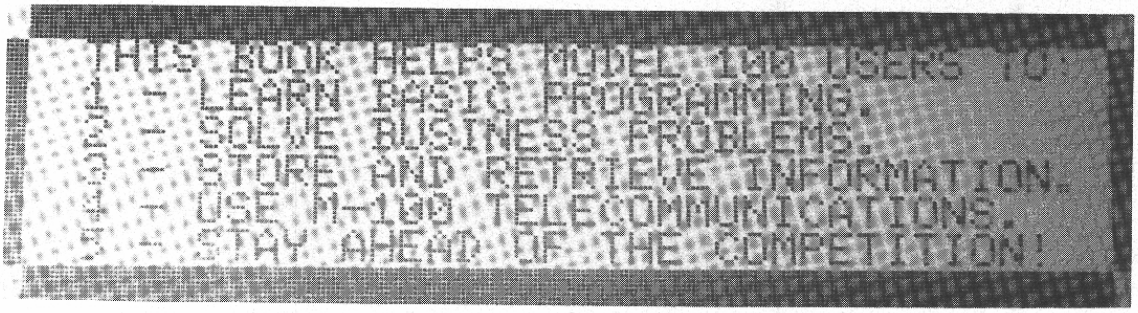


Figure 98 A Text Presentation

The first loop places CHR$(239), a black square, along the screen's top row. The second loop uses the same character to draw the border down the right-hand side of the screen. Notice that the STEP of the second loop is 40. This loop starts at the top of the screen, at position 79, and adds 40 to get to the position directly beneath it. The other loops work the same way, but by changing the loop parameters and the STEP they run the border across the bottom and up the left side of the screen, as shown in Figure 98.

Because there's a lot to read in this frame, the delay subroutine is called twice in line 450.

Fifth Frame

The loop at the beginning of the fifth frame again fills the screen with black. Then the PRINT @ statements and LINE commands, in lines 500–570, create a benefit flow chart, as shown in Figure

99. The color switch in the LINE commands is 0, to draw white lines, because the background is black. The BEEP command before each PRINT @ statement creates a short tone to emphasize each benefit.

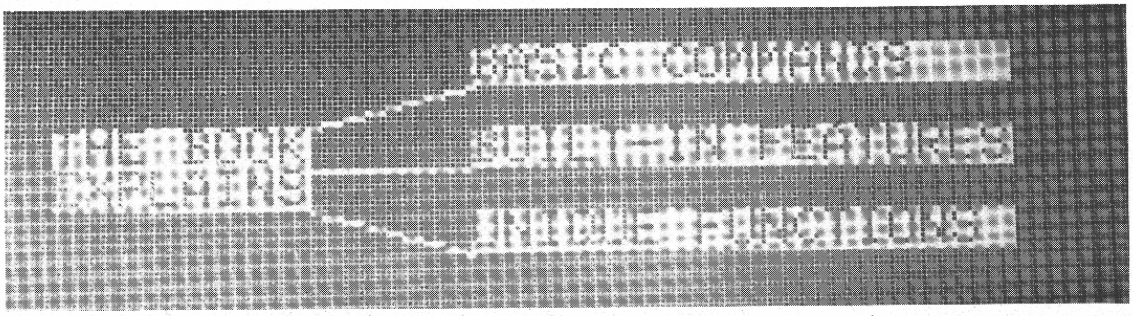


Figure 99 A Slide Presentation

Sixth Frame

Every good sales pitch asks the customer to use the product or service being sold. The sixth frame tells the viewer to "READ IT TODAY!" as shown in Figure 100. The graphics used are similar to the second frame, but the background is composed of CHR$(225), a gray tone. And the outline boxes use color switch 1, black.

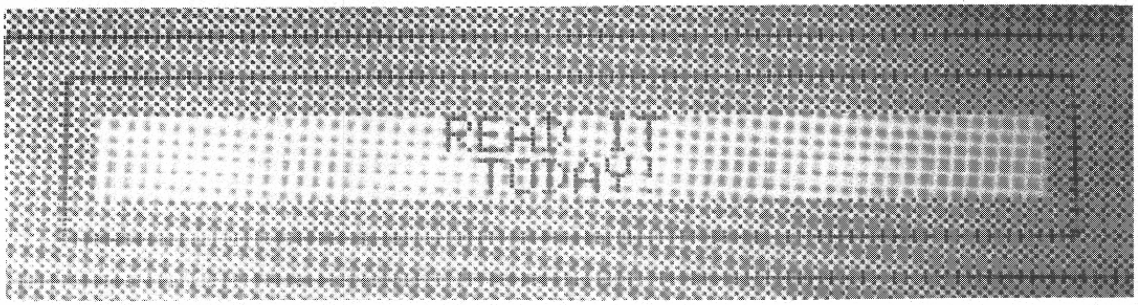


Figure 100 A Banner Using Graphics Characters

Line 690 sends the program back to line 30 to start over again.

Congratulations. You should be proud of yourself.
You've come a long way from the first commands in Chapter

0, and the Travel Expense Report program in Chapter 1. Now that you've read this far, and if you have tried to run all the programs in the book, you are well on your way to fluency in the Model 100 BASIC programming language.

When you write your own programs, you'll encounter bugs. Every program has them. The next chapter on program debugging can help you find the bugs in your programs and fix them.

Chapter 8

Getting the Bugs Out

When you write your own programs, or even when you type the sample programs from this book into your Model 100, you will discover something that millions of BASIC programmers have discovered before you: The first time you try to run a program, it does not do what you expect it to do. This could almost be stated as a universal law of computing—computer programs all have mistakes, called bugs. If you have never seen a computer program with a bug then you've never seen a computer program.

The first part of this chapter shows you how to find bugs in your programs. Then the second part of the chapter shows you how to use the Model 100's TEXT word processor to make program writing and debugging easier. First, we'll look at debugging.

There are two kinds of bugs. The first causes the program to crash. This is called a fatal error because the program dies. Ironically, this is the easiest type of bug to fix. When your computer program crashes, you at least get a hint about where the problem might reside.

The other type of bug is less obvious. While the fatal error sends the program into a tailspin, this type of bug can go *almost* undetected. When the program you write to compute your commission accepts your input, calculates the result, and prints out an elegant report, you may not notice if the calculation section you wrote is cheating you by a thousand dollars per month.

Between these two extremes are bugs caused by typographical errors and a long list of other mistakes that anyone can make. In fact, bugs should simply be called mistakes—because that's what they are.

The problem, then, isn't whether your programs will have bugs. The problem is finding and fixing them: a process known as program debugging.

Entire books have been written on debugging. This chapter introduces you to the basics by debugging the program shown in Figure 101. This Mailing List program should look familiar to you, for it's almost the same one that we used in Chapter 5.

```
10 'MAILING LIST PROGRAM USES THE DATA FROM
20 'ADRS.DO TO PRINT MAILING LABELS
30 CLS
40 PRNT "              MAILING LIST PROGRAM"
50 BEEP
60 PRINT:PRINT "PUT MAILING LABELS IN
PRINTER"
70 PRINT: LINE INPUT "PRESS 'ENTER' TO
CONTINUE...";X$
80 '--------------OPEN DATA FILE
90 OPEN "RAM:ADS.DO" FOR INPUT AS 1
100 IF EOF(1) THEN GOTO 230
110 '--------------GET DATA FROM FILE
120 INPUT #1, N$, CN$, ST$,CS$,ZP%,TN$
130 '--------------PRINT THE INFORMATION
ON LABEL
140 LPRINT N$
150 LPRINT CN$
160 PRINT ST$
170 LPRINT CS$;"   ";
180 LPRINT ZP%
190 LPRINT
200 LPRINT
210 '--------------REPEAT UNTIL ALL
LABELS PRINTED
220 GOTO 90
```

Figure 101 continues

```
230 CLOSE 1
240 CLS
250 PRINT "                    MAILING LIST
PROGRAM"
260 PRINT:PRINT "PRESS 'F4' TO RERUN"
270 END
```

Figure 101 Mailing List Program with Bugs

To make this debugging chapter challenging we have intentionally added a few mistakes to the program. This program uses the ADRS.DO file to print mailing labels. For the program to work properly, the ADRS.DO file must be like that shown in Figure 102.

```
Mr. S. Smith,A.B.C. Inc.,123 Main
St.,Anytown NY,12345,:2125551212:
Ms. J. Jones,Electro Corp.,111 Center
St.,Valley NY,12345,:2125551221:
Mr. N. Gineer,Computer Co.,21 Inter Ave.,
New City NY,12345, :2125551234:
Ms. A. Name,X.Y.Z. Inc.,11 Exec Way,
Central NY,12345,:2125554321:
```

Figure 102 Addresses Printout

To learn the debugging techniques type the program into your Model 100 exactly as it is in Figure 101 and debug it as shown below.

The first thing to do when you have to debug a program is to make a listing. A listing is a line by line printout of the program. There is nothing like having the printed listing in front of you, even if you can look at the program on the display screen. You can write on the listing, check off lines as you debug them, and draw arrows to debug program-flow problems.

Error Messages

When you try to run the version of the program in Figure 101 it will crash right away. The first thing you'll see displayed on the screen is this cryptic message:

```
?SN Error in 40
```

This is BASIC's way of telling you what is wrong with your program. It is called an error message. There are 33 error messages on the Model 100, listed in the Model 100 BASIC Error Codes chart in the owner's manual. They are also shown in the table in Figure 103.

Code	Message	Meaning
1	NF	NEXT without FOR.
2	SN	Syntax Error.
3	RG	RETURN without GOSUB.
4	OD	Out of Data.
5	FC	Illegal Function Call.
6	OV	Overflow.
7	OM	Out of Memory.
8	UL	Undefined Line.
9	BS	Bad Subscript.
10	DD	Doubly Dimensioned Array.
11	/0	Division by Zero.
12	ID	Illegal Direct.
13	TM	Type Mismatch.
14	OS	Out of String Space.
15	LS	String Too Long.
16	ST	String Formula Too Complex.
17	CN	Can't Continue.
18	IO	Input/Output Error.
19	NR	No RESUME.
20	RW	RESUME Without Error.
21	UE	Undefined Error.
22	MO	Missing Operand.
23–49	UE	Undefined Error.
50	IE	Internal Error.
51	BN	Bad File Number.
52	FF	File Not Found.
53	AO	Already Open.
54	EF	Input Past End of File.
55	NM	Bad File Name.
56	DS	Direct Statement in File.
57	FL	Undefined Error.
58	CF	File Not Open.
59–255	UE	Undefined Error.

Figure 103 Model 100 BASIC Error Codes

The Error Codes table contains three columns. The left column lists the error codes. These numbers are used to identify the error types. The center column lists the error messages (such as *SN*). The meanings are spelled out in the right-hand column. The error message:

```
?SN Error in 40
```

means that there is a syntax error in line 40. BASIC displays a syntax error message if there is a misspelled key word in a line of code, or if the command does not make sense to the interpreter. In line 40 of Figure 101 there is a typographical error in the key word PRINT. It is spelled PRNT.

```
40 PRNT "          MAILING LIST PROGRAM"
```

BASIC is very fussy about syntax. For the program to run, you have to correct this error. You could retype the entire line 40, but that is the hard way. The Model 100 has an excellent editor, the TEXT word processor. And one command, EDIT, moves your program automatically from BASIC command mode to the TEXT edit mode. You can then use all of the features of TEXT to correct the mistake.

EDIT

To enter TEXT from BASIC type:

EDIT (ENTER)

and press ENTER. The first eight lines of the program you are debugging will appear on the screen and the TEXT word-processing program will be functioning.

If you want to edit only the one line of the program, you can type the **EDIT** command with the line number:

EDIT 40 (ENTER)

and line **40** will appear on the screen. To edit a group of lines you can type:

EDIT 30-50 (ENTER)

Correcting Errors with TEXT

When you are in Edit mode use the arrow keys, or the Find function, to locate the syntax error and correct it. When you are finished correcting the syntax error in line 40, press key F8, Exit. The screen clears and the BASIC *Ok* message is displayed. This means that you are back in BASIC and can run the program to see if you fixed the problem.

When you try to Exit from TEXT you may see this message:

```
Text ill-formed
Press space bar for TEXT
```

This means that at least one line of the program is not a valid program line. BASIC does not know what to do with it and will not let you out of TEXT until you fix it.

To correct this condition press the space bar. The program will reappear on the screen and the cursor will be positioned at the line of code that needs fixing. Things to look for if you are stuck in TEXT are:

- A line of code without a line number
- A line of code longer than 255 characters
- An invalid line number (larger than 65529)
- A line of code that is only a line number. Every line number must be followed by at least the word REM or an apostrophe (').

More Errors

Once you fix the syntax error in line 40 and try to run the program, you will encounter another error, as shown in Figure 104.

```
MAILING LIST PROGRAM

PUT MAILING LABELS IN PRINTER

PRESS 'ENTER' TO CONTINUE...ADRS.DO
?FF Error in 90
Ok
```

Figure 104 File Not Found Error Message

```
?FF Error in 90
```

This File Not Found error message means that the file opened in line 90 was not found. It is not in the directory in the Model 100 main menu. Why is this? Look at line 90:

```
90 OPEN "RAM:ADS.DO" FOR INPUT AS 1
```

This bug is a little harder to find than the first. The filename, ADRS.DO, is misspelled. This is really another typographical error, but BASIC only knows that it can't find the file ADS.DO. It displays the File Not Found message. You can't argue if you see it from BASIC's point of view. To fix line 90 type:

```
EDIT 90
```

and change the filename in the OPEN command to ADRS.DO.

Subtle Bugs

Bugs in computer programs can be subtle and confounding. After you fix the bug in line 90 and try to run the program, it will print one mailing label and then crash again. The screen will look like Figure 105.

```
PUT MAILING LABELS IN PRINTER

PRESS 'ENTER' TO CONTINUE...ADRS.DO
123 Main St.
?AO Error in 90
Ok
```

Figure 105 File Already Open Error Message

The error message is displayed along with something else:

```
123 Main St.
?AO Error in 90
```

This is frustrating for two reasons. First, we already fixed line 90 and it looks as though there is nothing else wrong with it. Second, what is *123 Main St.* doing on the display screen? Isn't this supposed to be part of the mailing label?

To debug subtle problems like this remember that a BASIC program can only do one thing at a time, one line of code at a time. Therefore a way to find the bug is to go through the program one line at a time as if you were the BASIC interpreter. Don't do what you expect the program to do. Do what the program is telling the interpreter to do.

At first glance, things look O.K. Line 100 checks for end of file. That's not causing our problem. Line 120 gets the data from the file, and that isn't crashing the program. Lines 140–180 print the mailing label. Since one of these lines prints the street address, 123 Main St., part of the problem might be here. And it is, in line 160:

```
160 PRINT ST$
```

This line should have an LPRINT, not a PRINT:

```
160 LPRINT ST$
```

The street address was sent to the display screen instead of the printer. That solves half the problem. But what about the error message, ?AO Error in 90?

An ?AO error, Already Open, means that the program is instructing BASIC to OPEN a file that is already open. Lines 170–200 are not the offenders. What about line 220?

```
220 GOTO 90
```

When line 220 sends the program back to the top of the loop for more mailing label data, it overshoots the mark. When it executes line 90 it tries to open the already open ADRS.DO file and crashes. To fix this, change line 220 to read:

```
220 GOTO 100
```

The program is now debugged. If your ADRS.DO file was like Figure 102 the output should look like Figure 61, Chapter 5.

ON ERROR Interrupt

Some errors can be anticipated and steps taken to avoid program crashes. When the user is asked to enter information at the keyboard, there is always a chance that something will go wrong. In the Order Entry program, Figure 62, the user is asked in line 580 for the name of the:

```
FILE TO PRINT?...
```

The program displays a directory of files in RAM, as shown in Figure 69. If the user entered MIAN.DO, MANI.DO, NAIN.DO or any of the numerous typographical errors that are possible, the program will crash and display a File Not Found error message. But what is the average Model 100 user going to do with an ?FF Error in 590 message?

One way to avoid this is with the **ON ERROR** interrupt. The **ON ERROR** interrupt follows the form:

ON ERROR GOTO *line number*

where the line number is the beginning of a subroutine that handles the error. A subroutine that handles an error is sometimes called an error-trapping subroutine. The Order Entry program in Figure 62 could be modified with the following error-trapping subroutine shown in Figure 105.

```
5 ON ERROR GOTO 1000
"
"          PROGRAM LINES 10-490
"
1000     'ERROR TRAPPING SUBROUTINE
1010 IF ERR = 52 THEN CLS ELSE GOTO 1080
1020 PRINT "FILE ";FN$;" NOT FOUND"
1030 PRINT "BE SURE THE FILE TO PRINT"
1040 PRINT "IS LISTED IN DIRECTORY."
1050 PRINT
1060 LINE INPUT "PRESS 'ENTER' TO
CONTINUE...";X$
1070 RESUME 560
1080      'PRINT ERROR CODE OTHER THAN FILE
NOT FOUND
1090 PRINT "ERROR CODE ";ERR;" IN LINE
";ERL
1100 RESUME NEXT
```

Figure 106 Error Trapping Subroutine

To use an error-trapping subroutine with the Model 100 the first line of the program should be like this:

```
05 ON ERROR GOTO 1000
```

This alerts BASIC to jump to the subroutine in lines 1000–1070 when an error occurs. Because the program jumps to a subroutine, you would expect the command to be ON ERROR GOSUB but it is not. I don't know why not, that's just the way Model 100 BASIC was designed.

Let's see how the subroutine traps the error. Line 1010 checks to see if the File Not Found error (error code 52) has occurred. In line 1010 *ERR* is Model 100's variable for the error code. If error 52 (File Not Found) has occurred the screen is cleared and the following message is displayed:

```
FILE filename NOT FOUND
BE SURE THE FILE TO PRINT
IS LISTED IN DIRECTORY

PRESS 'ENTER' TO CONTINUE...
```

When the user presses ENTER, line 1070 returns the program to line 560. The file directory is displayed again and the user can reenter the name of the file to be printed. **RESUME** is a special type of RETURN, used at the end of error-trapping subroutines. There are two forms:

RESUME NEXT

returns the program to the line following the line that caused the error.

RESUME *line number*

returns the program to the *line number* that is specified.

If the error was found not to be error 52 in line 1010 then the subroutine jumps to line 1080. In line 1090 the error code is displayed. **ERL,** used at the end of line 1090, is the Model 100's variable for the **error line.**

```
1090 PRINT "ERROR CODE";ERR;" IN
LINE ;ERL
```

Line 1100 returns control to the line after the error line.

Writing Programs With TEXT

The TEXT word processor is ideal for writing programs, not only debugging them. When you write programs you can use all of

TEXT's features, such as FIND, CUT, COPY, and PASTE, to help you save time. And if you are using the Model 100 for business, time may be your most valuable resource.

The document that you create when you write a computer program—the line numbers and BASIC commands—is called the source code. It is called source code because it's the source of instructions that the interpreter will translate into machine code for the Model 100.

It is easier to write source code with a word processor than it is in BASIC command mode. And in the Model 100 all you have to do to test the program is press key F8 to leave TEXT and go to the main menu, enter BASIC command mode by placing the cursor over the program's filename, and then press ENTER to run the program. Easy.

If you make a minor error writing a line of code in command mode you have to retype the entire line. If it is a complex command, such as an IF . . . THEN statement or an OPEN command, retyping the line can be a real waste of time. When you use TEXT to write programs you can correct typographical errors as you write each line of the program because with TEXT you can use the cursor controls to move anywhere in the document. You can insert characters, words, or entire lines of code.

With TEXT you can recycle modules of programs that you have successfully written and debugged. There is no sense in reinventing the wheel every time you write a program. If you have a good data-entry routine, you can move it from the orginal program using TEXT's **COPY** and **PASTE** commands (described in the Model 100 owner's manual under TEXT). Then all you have to do is renumber the lines and make any other changes to tailor the routine to your new program. Subroutines can often be recycled in this way because they perform functions, such as getting input from the user or sending output to the printer, used in most programs.

COPY and **PASTE** can also save you time and help prevent typographical errors. If you are writing numerous LINE INPUT statements you can **COPY** the beginning phrase:

```
LINE INPUT "
```

Then after typing the line number, you simply press the **PASTE** command key and position the cursor to type the message between the quotation marks. **COPY** and **PASTE** can be especially helpful when writing cumbersome lines of code such as PRINT USING statements. Why retype a line like:

```
10 PRINT USING "$$$####.##";
```

when you can just press the PASTE key and type the variable name?

TEXT's **FIND** can help if you have to renumber program lines. All the program lines in this book are numbered in even steps of ten, and they look great. But when you write programs you almost always have to insert a line of code here and there. If you want to renumber lines in nice increments of ten, then you have to renumber all the GOTO and GOSUB commands. You can use **FIND** to locate all the GOTOs and GOSUBs and change the numbers to which they refer. **FIND** can also be used to locate all the locations of a variable if you have to change the variable name or variable type.

Chapter 9

Model 100 Disk-BASIC

This chapter provides the information that you will need to use the Disk/Video Interface with Disk-BASIC to create and use data files, and to save and load program files to and from diskette.

Disk-BASIC operations are similar to those used by Model 100 BASIC to create and use RAM and cassette files. The fundamental techniques of file creation and use are covered in Chapter 5, so you should be familiar with the information in Chapter 5 before you start this chapter.

In addition, this chapter moves very rapidly because almost all the commands in Disk-BASIC are duplicates of commands that we have already covered. For example, there are a SAVE and a LOAD command for Disk-BASIC. With very minor changes, these Disk-BASIC commands are identical to the Model 100-BASIC SAVE and LOAD commands. Let's get started.

Why Disk/Video Interface?

The 5¼-inch floppy diskettes (also known as diskettes, or just "floppies") are the most popular form of data storage for micro-computers. When compared with cassette tape, diskettes have many advantages. Data can be recorded faster on floppy disks than

on cassette tape, and the disk-drive unit can move very rapidly to read or record information anywhere on the diskette. Diskettes can be copied quickly. Diskettes hold more data than cassette tapes. And even though no form of magnetic storage is 100 percent secure, diskettes can be more reliable than tape.

To allow you to use diskettes with the Model 100, Radio Shack sells the optional Model 100 Disk/Video Interface. That's Radio Shack's name for a unit that contains a 5¼-inch-floppy-disk drive unit, a floppy disk interface, a video interface, and cables to connect the unit to the Model 100. An interface is jargon for the electronic and electro-mechanical parts that allow you to connect two electronic units. The floppy disk interface connects the Model 100 to the floppy disk drive. The video interface connects the Model 100 to a video monitor or home television set. The Disk/Video interface unit can be expanded by the addition of one more disk-drive unit, for a maximum of two disk drives.

The Disk/Video Interface package includes software. The software is shipped on a diskette, called the System Diskette. The System Diskette contains three types of software for use with Disk/Video Interface:

1. The Portable Computer Disk Operating System (DOS). The Portable Computer DOS is a computer program that manages the operation of the disk drive unit and the video monitor.
2. The utility programs, FORMAT, BACKUP.SNG (SNG stands for SiNGle. BACKUP.SNG is the backup program to use if you only have one disk drive), and BACKUP. FORMAT prepares diskettes for use. The BACKUP and BACKUP.SNG programs copy data and "system tracks" onto diskettes.
3. The expanded version of BASIC, called Disk-BASIC. Disk-BASIC used Model 100 BASIC as a foundation and adds commands for controlling the disk operations.

Starting the System and Loading Disk-BASIC

Connect the Disk/Video Interface as shown in the Disk/Video Interface owner's manual. To avoid major problems, be sure to follow all the instructions in the owner's manual when you connect or disconnect the Disk/Video Interface and when you turn the Model 100 or the Disk-Video Interface power on or off.

Before you can load Disk-BASIC into the Model 100 you must load Portable Computer DOS into the Disk/Video Interface according to the instructions in the Disk/Video Interface owner's manual. The programs in DOS are stored in the RAM of the Disk/Video Interface.

Disk-BASIC is contained on the diskette called the System Diskette. Before you can use Disk-BASIC you have to transfer it from the System Diskette into the RAM of the Model 100 using one of two methods, warm start or cold start. You will need at least 4500 free bytes of Model 100 RAM to load and store Disk-BASIC.

Warm Start

To perform a warm start turn the power switch off and then on again.

Cold Start

> IMPORTANT NOTE: When you use the cold start method, *all the programs in the Model 100 RAM are erased forever*. If you choose to use the cold start method, *back up all the programs in RAM to cassette tape before proceeding*. If you do not make backups of all programs in RAM, they will be lost *permanently*.

To perform a cold start, press and hold the CTRL and PAUSE keys, and the RESET switch, in that order. Release the RESET switch but continue to hold down the CTRL and PAUSE keys. When the main menu appears on the LCD screen release the CTRL and PAUSE keys.

Disk-BASIC will stay in the RAM of the Model 100 even after you turn off the power, but it will not be listed as a file in the main menu. All that is listed in the menu are BASIC and the built-in programs, as they were when you first turned on the Model 100.

Preparing Diskettes With FORMAT

Data is stored on the magnetic coating of the diskette in concentric circles called tracks. The tracks are divided into arcs, called sectors. The tracks and sectors are used by the Portable Computer DOS to keep track of the data on the diskette. The illustration in Figure 107 is a general representation of the way that tracks and sectors are arranged on a diskette. The FORMAT program is used to electronically mark the beginning of the tracks and sectors on the diskette according to the plan recognized by DOS. FORMAT also checks for flaws in the diskette.

Before diskettes can be used, they must be prepared with the FORMAT program as shown in the owner's manual. This is called formatting or initializing the diskette. Formatting also erases the diskette. All data and programs on the diskette, if any, will be wiped out permanently when the diskette is formatted.

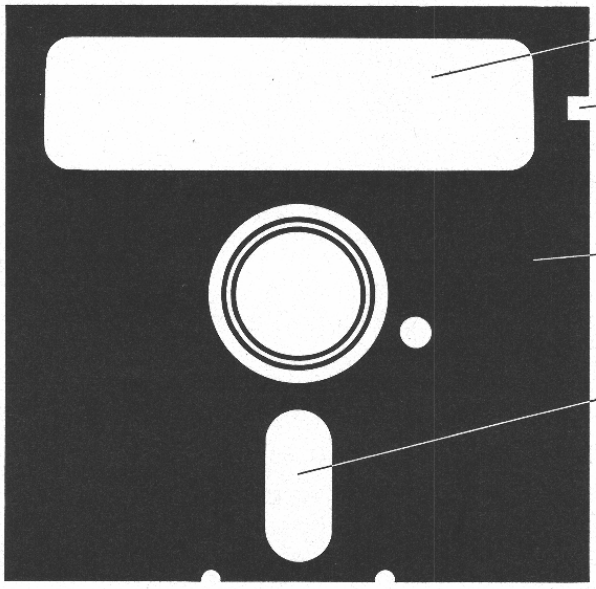
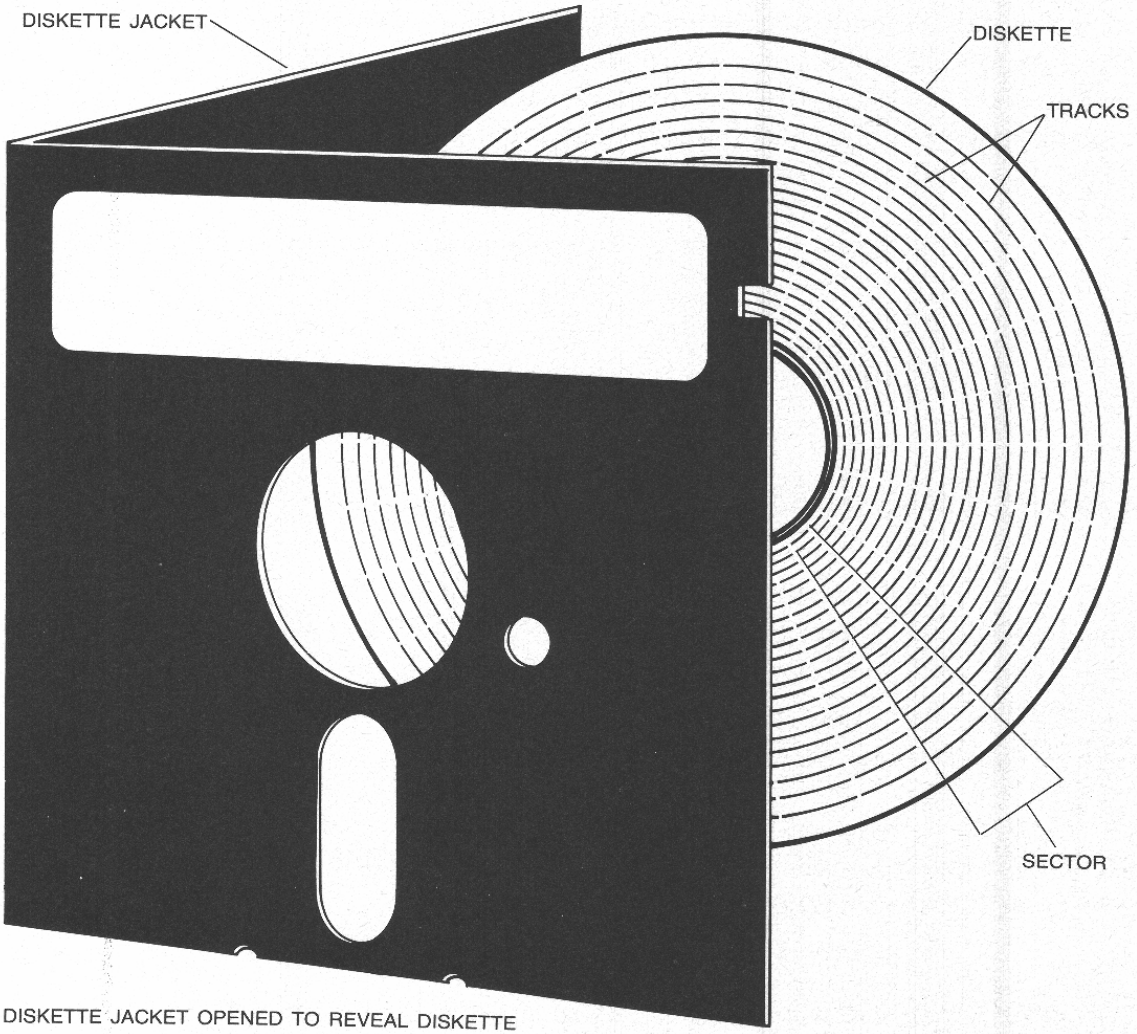


Figure 107 continues



DISKETTE JACKET OPENED TO REVEAL DISKETTE

Figure 107 Anatomy of a Diskette

Disk-BASIC Filenames

Filenames in Disk-BASIC are different from those in BASIC. A Disk-BASIC filename must begin with a letter and consist of up to six alphanumeric characters. The extension consists of three characters. The file extension is treated as part of the filename in Disk-BASIC.

Disk-BASIC recognizes three types of files: BASIC program files, ASCII files, and Machine Language program files. Disk-BASIC

separates the filename and the extension with a "punctuation" character to indicate the file type. BASIC program files are indicated by a period (.) between the filename and extension, ASCII files by a blank space ( ), and Machine Language program files by an asterisk (*), as shown below:

File Type	Designation
BASIC	FLNAME.EXT
ASCII	FLNAME EXT
Machine Language	FLNAME*EXT

Using Disk-BASIC

When you use files in Disk-BASIC programs, you have to specify the disk drive that contains the file. Portable Computer DOS calls the first disk drive 0 and the second optional disk drive 1 (referred to as drive zero and drive one). You can load programs from a diskette and save programs onto a diskette, just as you could load and save programs to and from RAM or tape.

SAVE

To save the current BASIC program on diskette, use the disk-BASIC version of the SAVE command:

SAVE "*drive number*:FLNAME"

The *drive number* is either 0 or 1. To save the program named GAME on a diskette in drive zero type:

SAVE "0:GAME"

Press ENTER. The disk-drive light will go on, and the disk will spin as the file is stored on the diskette.

To save the same program on a diskette in the optional drive one type:

SAVE "1:GAME"

and press ENTER.

LOAD

To load a BASIC program from a diskette to the Model 100 RAM use the disk-BASIC version of the LOAD command:

LOAD "*drive number*:FLNAME"

For example, to load the program named SALES, from drive zero to RAM type:

LOAD "0:SALES"

Press ENTER. DOS will read the program from the diskette and send it to the Model 100.

LFILES

Do you recall from Chapter 5 that you can display a directory of the files stored in RAM by typing the FILES command? A directory is simply a list of filenames. The FILES command follows the form:

FILES (ENTER)

To display a directory of files on a diskette use the **LFILES** command. It follows the form:

LFILES *drive number*

The *drive number* is either 0 or 1. To see a directory of files on drive zero type:

LFILES 0 (ENTER)

The following information is then displayed:

```
SYSTEM VER 01.00.00
FORMAT.    1    BACKUP. 2
BACKUP.SNG 2
155.25 K AVAILABLE
```

The number on the last line of the display is the amount of memory available on the diskette, in kilobytes. In the example above there are 155.25 K available.

KILL

To delete a file from the diskette use the Disk-BASIC **KILL** command:

KILL "*drive number*:FLNAME.EXT"

The command:

KILL "0:TRIG"

deletes the file TRIG from the diskette in drive zero. The **KILL** command deletes the file *permanently*.

NAME

To rename a file, use the Disk-BASIC version of the **NAME** command. It takes the form:

NAME "*drive #*:FLNAME.EXT" AS "*drive #*:NUNAME.EXT"

FLNAME.EXT is the old filename, NUNAME.EXT is the new filename.

RUN

You can load a file from a diskette and run it by using the **RUN** command:

RUN "*drive number*:FLNAME.EXT"

Enter the command:

RUN "1:STOCK"

and the program STOCK is loaded from the diskette in drive one into the Model 100 memory. It then **RUNS.**

SCREEN and WIDTH

With the Disk/Video Interface you can also attach a video monitor to display the Model 100's screen output.

SCREEN lets you switch the display from the Model 100 LCD

screen to the video monitor attached to the Disk/Video Interface. The SCREEN command lets you tell Disk-BASIC that either the LCD screen or the video monitor is attached to the Disk/Video Interface. If you enter the command:

SCREEN 0,0 (ENTER)

the display is sent to the LCD screen. While the command:

SCREEN 1,0 (ENTER)

sends the display to the video screen.

The **WIDTH** command sets the width of the display on the CRT screen to either 40 or 80. **WIDTH** follows the form:

WIDTH *screen width*

Screen width can only be either 40 or 80. If you are using a home TV you will probably want to set the WIDTH to 40:

WIDTH 40 (ENTER)

If you have a video monitor, designed especially for use with computers, it may be able to display 80 characters per line. In that case set the WIDTH to 80:

WIDTH 80 (ENTER)

Creating and Using Files with Disk-BASIC

Disk-BASIC allows you to create and use sequential files on diskettes. The Disk-BASIC commands used to manipulate files are very similar to those used by Model 100 BASIC to create sequential RAM files.

The fundamental disk file operations are controlled by the Disk-BASIC statements listed below. Notice that they perform functions that correspond to their Model 100 BASIC equivalents.

Statement	Function Performed
OPEN	Opens a disk file
CLOSE	Closes a disk file
INPUT #	Reads data from disk sequentially
PRINT #	Records data on disk sequentially

When converting a program from Model 100 BASIC to Disk-BASIC the principal instructions that are different are the OPEN statements. The Disk-BASIC OPEN statement follows the form:

OPEN *"drive # : filename"* FOR *mode* AS *file number*

The filename, mode, and file number function as they do in Model 100 BASIC.

- The filename is the name of the file to be opened.
- The mode is either OUTPUT or INPUT. Diskette files are opened FOR OUTPUT when you want the program to store data. Diskette files are opened FOR INPUT when you want the program to retrieve data.
- The file number is usually 1. If a second file is opened it will be 2, the third open file will be 3, and so on.

The drive number (new to Disk-BASIC) is either 0 or 1. For example, to open a data file named DATA on a diskette in drive zero the Disk-BASIC instruction would be:

```
10 OPEN "0:DATA" FOR OUTPUT AS 1
```

The program in Figure 108 is an adaptation of the program in Figure 50. Notice that the only changes that have been made to the program are in lines 90 and 190, the **OPEN** statements. This program is thoroughly explained in Chapter 5.

```
10 'USING A SEQUENTIAL DISK DATA FILE
20 'STEP 1:
30 'GET DATA FROM USER
40 CLS
```

Figure 108 continues

```
50 LINE INPUT "ENTER NAME...";NM$
60 INPUT "ENTER PRODUCT CODE...";PC
70 'STEP 2:
80 'OPEN SEQUENTIAL DISK FILE FOR STORAGE
OF DATA
90 OPEN "0:DATA" FOR OUTPUT AS 1
100  'STEP 3:
110  'STORE DATA IN DISK FILE
120 PRINT #1, NM$;",";PC
130 'STEP 4
140 'CLOSE DISK DATA FILE
150 CLOSE 1
160 CLS: LINE INPUT "PRESS 'ENTER' TO
RETRIEVE DATA...";X$
170 'STEP 5:
180 'OPEN SEQUENTIAL DISK FILE TO RETRIEVE
DATA
190 OPEN "0:DATA" FOR INPUT AS 1
200 'STEP 6
210 'RETRIEVE DATA FROM DISK DATA FILE
220 INPUT #1, NM$, PC
230 'STEP 7
240 'CLOSE DISK DATA FILE
250 CLOSE 1
260 'STEP 8
270 'USE RETRIEVED DATA IN CALCULATION OR
OUTPUT
280 CLS
290 PRINT "HERE IS YOUR DATA..."
300 PRINT :PRINT "THE CUSTOMER - ";NM$
310 PRINT "USES PRODUCT NUMBER";PC
320 END
```

Figure 108 Using a Sequential Disk Data File with Disk-BASIC

Line 90 was:

```
90 OPEN "RAM:DATA.DO" FOR APPEND AS 1
```

To open a *disk* data file line 90 is changed to read:

```
90 OPEN "0:DATA" FOR OUTPUT AS 1
```

The drive number replaces the device specification RAM. **OUTPUT** is used instead of **APPEND.**

Searching for Data in a Sequential Disk File

To open a disk data file and search for data you must use an algorithm that checks for the end of the file (EOF) before reading each data item.

Suppose that a sequential disk data file of customer information, named TELDAT, contains customer names and telephone numbers. The data is stored in the TELDAT data file like this:

ABC CORP., 2125551234, XYZ Inc., 2125551232
APEX Co., 2125554321, and so on . . .

The routine below opens a disk data file and searches for the name of the customer, XYZ Inc. (This example assumes that the user enters "XYZ Inc." in response to the line input statement in line 290.) When the program finds XYZ Inc. it prints the name and phone number information on the screen and closes the file. The variable for the company name is CN$. The variable for the phone number is PH$.

```
.
.  previous lines of code
.
290 LINE INPUT "Enter company name
. . ."; X$
300 OPEN "0:TELDAT" FOR INPUT AS 1
310 IF EOF(1) THEN GOTO 370
320 INPUT #1, CN$, PH$
330 IF CN$ <> X$ THEN GOTO 310
340 CLS
```

```
350 PRINT CN$
360 PRINT "Phone number ";PH$ :GOTO 390
370 CLS
380 PRINT X$;"not found."
390 CLOSE 1
.
.  succeeding lines of code
.
```

The main difference between the routine shown above and those used with RAM data files is the **OPEN** statement in line 300. This data search routine is thoroughly explained in Chapter 5.

Data Security When Using Diskettes

Here are some steps to follow for maximum data security when using diskettes.

- Always make backup copies of diskettes. *No* data storage medium is 100 percent secure. Make backups at least once per day and more often when you have data that is worth protecting. Use the BACKUP or BACKUP.SNG utility programs following the instructions in the owner's manual. Print hard copies of important data, too.

- Beware of magnets. A diskette is just a flat sheet of plastic coated with the same material used to coat magnetic tape. As with tape, magnetic fields will erase data from the diskette. In offices magnets are often found on copy stands and paper clip holders. Paper clips stored in these little holders become magnetized. Using one to hold a piece of paper to a diskette is asking for trouble.

  Magnetic fields are created by electric current running through wires, motors, and electronic devices, including power cords, electric appliances, TVs, radios, and loud speakers. Don't store diskettes on top of or near these devices, not even for a moment.

- Never store a diskette in the disk drive. Do not insert a diskette in the drive before turning the power on. Always remove dis-

kettes from the disk drive unit before turning the power off. When the power is turned on and off the change in electrical current flowing to the disk drives can make them "record," wiping out the data on your diskettes.

- Handle diskettes carefully by the jacket only. Do not touch the exposed shiny surfaces of the magnetic coating. Do not bend the disk. When writing on the disk label, use only a soft felt-tipped marker. The surface of the diskette where your valuable data is stored is very delicate.
- Store disks vertically in boxes designed for the purpose. Store disks in their protective sleeves so they don't become contaminated.

Disk-BASIC Error Codes

The table in Figure 109 shows the error codes and error messages for Disk-Basic. This table contains new error codes and messages, such as 60, *Bad Drive Number;* 61, *Bad Track/Sector;* and 63, *Disk Full,* that are unique to Disk-BASIC.

The Bad Drive Number error message **(DN)** will be displayed if a command or file name refers to a drive that does not exist. For example, if you do not have the optional Drive One and a program tries to execute the statement:

```
OPEN "1:FLNAME" FOR OUTPUT AS 1
```

this error message will be displayed: **DN** error. **DN** stands for bad Drive Number.

The Bad Track/Sector error message **(TS)** indicates a damaged track or sector on the diskette. This can be one of the most frustrating error messages to see because it often means that data stored on the bad diskette is lost. DOS will attempt to write to or read from the track or sector, but if it is unsuccessful this error message is displayed.

The Disk Full error message **(DF)** is self-explanatory. As part of routine management of the Model 100 system, use the LFILES command to determine the amount of storage space left on your diskettes.

Code	Message	Meaning
1	NF	NEXT Without FOR.
2	SN	Syntax Error.
3	RG	RETURN Without GOSUB.
4	OD	Out of Data.
5	FC	Illegal Function Call.
6	OV	Overflow.
7	OM	Out of Memory.
8	UL	Undefined Line.
9	BS	Bad Subscript.
10	DD	Double Dimensioned Array.
11	/0	Division by Zero.
12	ID	Illegal Direct.
13	TM	Type Mismatch.
14	OS	Out of String Search.
15	LS	String Too Long.
16	ST	String Formula Too Complex.
17	CN	Can't Continue.
18	IO	I/O Error.
19	NR	No RESUME.
20	RW	RESUME Without Error.
21	UE	Undefined Error.
22	MO	Missing Operand.
23–49	UE	Undefined Error.
50	IE	Internal Error.
51	BN	Bad File Number.
52	FF	File Not Found.
53	AO	Already Open.
54	EF	Input Past End of File.
55	NM	Bad Filename.
56	DS	Direct Statement in File.
57	FL	Too Many Files.
58	CF	File Not Open.
59	AT	Bad Allocation Table.
60	DN	Bad Drive Number.
61	TS	Bad Track/Sector.
62	FE	File Already Exists.
63	DF	Disk Full.
64–255	UE	Undefined Error.

Figure 109 Disk-BASIC Error Codes

And now you know the BASIC programming language. As we said in Chapter 0, with the Model 100 and your fast-track spirit we're sure you'll be writing your own programs. Don't forget the seven rules of programming. They're listed in Appendix B, just in case you need them. And don't forget that bugs are the best thing that can happen when you're trying to learn programming. If you have trouble finding a bug in your programs, refer back to Chapter 8. Good luck, and happy programming.

Appendix A

ASCII Character Code Tables

Decimal	Hex	Binary	Printed Character	Keyboard Character
0	00	00000000		PAUSE
1	01	00000001		CTRL A
2	02	00000010		CTRL B
3	03	00000011		CTRL C
4	04	00000100		CTRL D
5	05	00000101		CTRL E
6	06	00000110		CTRL F
7	07	00000111		CTRL G
8	08	00001000		CTRL H
9	09	00001001		CTRL I
10	0A	00001010		CTRL J
11	0B	00001011		CTRL K
12	0C	00001100		CTRL L
13	0D	00001101		CTRL M
14	0E	00001110		CTRL N
15	0F	00001111		CTRL O
16	10	00010000		CTRL P
17	11	00010001		CTRL Q
18	12	00010010		CTRL R
19	13	00010011		CTRL S
20	14	00010100		CTRL T
21	15	00010101		CTRL U
22	16	00010110		CTRL V
23	17	00010111		CTRL W
24	18	00011000		CTRL X
25	19	00011001		CTRL Y
26	1A	00011010		CTRL Z
27	1B	00011011		ESC
28	1C	00011100		→
29	1D	00011101		←
30	1E	00011110		↑
31	1F	00011111		↓
32	20	00100000		SPACEBAR
33	21	00100001	!	!
34	22	00100010	"	"
35	23	00100011	#	#
36	24	00100100	$	$
37	25	00100101	%	%
38	26	00100110	&	&
39	27	00100111	'	'
40	28	00101000	(	(
41	29	00101001	)	)
42	2A	00101010	*	*
43	2B	00101011	+	+
44	2C	00101100	,	,
45	2D	00101101	–	–
46	2E	00101110	.	.
47	2F	00101111		
48	30	00110000	0	0
49	31	00110001	1	1
50	32	00110010	2	2
51	33	00110011	3	3
52	34	00110100	4	4
53	35	00110101	5	5
54	36	00110110	6	6
55	37	00110111	7	7
56	38	00111000	8	8
57	39	00111001	9	9
58	3A	00111010	:	:
59	3B	00111011	;	;

Decimal	Hex	Binary	Printed Character	Keyboard Character
60	3C	00111100	<	<
61	3D	00111101	=	=
62	3E	00111110	>	>
63	3F	00111111	?	?
64	40	01000000	@	@
65	**41**	**01000001**	**A**	**A**
66	42	01000010	B	B
67	43	01000011	C	C
68	44	01000100	D	D
69	45	01000101	E	E
70	46	01000110	F	F
71	47	01000111	G	G
72	48	01001000	H	H
73	49	01001001	I	I
74	4A	01001010	J	J
75	4B	01001011	K	K
76	4C	01001100	L	L
77	4D	01001101	M	M
78	4E	01001110	N	N
79	4F	01001111	O	O
80	50	01010000	P	P
81	51	01010001	Q	Q
82	52	01010010	R	R
83	53	01010011	S	S
84	54	01010100	T	T
85	55	01010101	U	U
86	56	01010110	V	V
87	57	01010111	W	W
88	58	01011000	X	X
89	59	01011001	Y	Y
90	5A	01011010	Z	Z
91	5B	01011011	[	[
92	5C	01011100	\	GRPH –
93	5D	01011101	]	]
94	5E	01011110	^	^
95	5F	01011111	_	_
96	60	01100000	`	GRAPH [
97	61	01100001	a	A
98	62	01100010	b	B
99	63	01100011	c	C
100	64	01100100	d	D
101	65	01100101	e	E
102	66	01100110	f	F

* For uppercase letters A-Z, press SHIFT or CAPS LOCK before pressing the Keyboard Character.

Decimal	Hex	Binary	Printed Character	Keyboard Character
103	67	01100111	g	G
104	68	01101000	h	H
105	69	01101001	i	I
106	6A	01101010	j	J
107	6B	01101011	k	K
108	6C	01101100	l	L
109	6D	01101101	m	M
110	6E	01101110	n	N
111	6F	01101111	o	O
112	70	01110000	p	P
113	71	01110001	q	Q
114	72	01110010	r	R
115	73	01110011	s	S
116	74	01110100	t	T
117	75	01110101	u	U
118	76	01110110	v	V
119	77	01110111	w	W
120	78	01111000	x	X
121	79	01111001	y	Y
122	7A	01111010	z	Z
123	7B	01111011	{	GRPH 9
124	7C	01111100	|	GRPH _
125	7D	01111101	}	GRPH 0
126	7E	01111110	~	GRPH |
127	7F	01111111		DEL
128	80	10000000	☎	GRPH p
129	81	10000001	⍐	GRPH m
130	82	10000010	(×	GRPH f
131	83	10000011	◤	GRPH x
132	84	10000100	⌗	GRPH c
133	85	10000101	⍏	GRPH a
134	86	10000110	⍓	GRPH h
135	87	10000111	⍑	GRPH t
136	88	10001000	⍐	GRPH l
137	89	10001001	\ ¯	GRPH r
138	8A	10001010	≠	GRPH /
139	8B	10001011	Σ	GRPH s
140	8C	10001100	≈	GRPH '
141	8D	10001101	÷	GRPH =
142	BE	10001110	∫	GRPH i
143	BF	10001111	◀	GRPH e
144	90	10010000	⍓	GRPH y
145	91	10010001	⍗	GRPH u

* For lowercase letters a-z, be sure CAPS LOCK is not pressed "down."

Decimal	Hex	Binary	Printed Character	Keyboard Character
146	92	10010010	↕	GRAPH ;
147	93	10010011	웃	GRAPH q
148	94	10010100	웃	GRAPH w
149	95	10010101	♂	GRAPH b
150	96	10010110	♀	GRAPH n
151	97	10010111	⁄	GRAPH .
152	98	10011000	▮	GRAPH o
153	99	10011001	▮	GRAPH ,
154	9A	10011010	→	GRAPH l
155	9B	10011011	←	GRAPH k
156	9C	10011100	≠	GRAPH 2
157	9D	10011101	≤	GRAPH 3
158	9E	10011110	≥	GRAPH 4
159	9F	10011111	◇	GRAPH 5
160	A0	10100000	·	CODE '
161	A1	10100001	à	CODE x
162	A2	10100010	ç	CODE c
163	A3	10100011	£	GRAPH 8
164	A4	10100100	`	CODE "
165	A5	10100101	μ	CODE M
166	A6	10100110		CODE)
167	A7	10100111	▼	CODE _
168	A8	10101000	†	CODE +
169	A9	10101001	§	CODE s
170	AA	10101010	R	CODE R
171	AB	10101011	C	CODE C
172	AC	10101100	¼	CODE p
173	AD	10101101	¾	CODE ;
174	AE	10101110	½	CODE /
175	AF	10101111	¶	CODE 0
176	B0	10110000	¥	GRAPH 7
177	B1	10110001	Ä	CODE A
178	B2	10110010	Ö	CODE O
179	B3	10110011	Ü	CODE U
180	B4	10110100	¢	GRAPH 6
181	B5	10110101	~	CODE |
182	B6	10110110	ä	CODE a
183	B7	10110111	ö	CODE o
184	B8	10111000	ü	CODE u
185	B9	10111001	B	CODE S
186	BA	10111010	TM	CODE T
187	BB	10111011	é	CODE d
188	BC	10111100	ù	CODE ,
189	BD	10111101	è	CODE v
190	BE	10111110	¨	CODE =

Decimal	Hex	Binary	Printed Character	Keyboard Character
191	BF	10111111	£	CODE F
192	C0	11000000	â	CODE I
193	C1	11000001	ê	CODE 3
194	C2	11000010	î	CODE 8
195	C3	11000011	ô	CODE 9
196	C4	11000100	û	CODE 7
197	C5	11000101		CODE –
198	C6	11000110	ë	CODE e
199	C7	11000111	ï	CODE i
200	C8	11001000	á	CODE q
201	C9	11001001	í	CODE k
202	CA	11001010	ó	CODE l
203	CB	11001011	ú	CODE j
204	CC	11001100	ý	CODE y
205	CD	11001101	ñ	CODE n
206	CE	11001110	â	CODE z
207	CF	11001111	õ	CODE .
208	D0	11010000	À	CODE !
209	D1	11010001	È	CODE #
210	D2	11010010	Ì	CODE *
211	D3	11010011	Ò	CODE (
212	D4	11010100	Ù	CODE &
213	D5	11010101	Ï	CODE I
214	D6	11010110	Ë	CODE E
215	D7	11010111	É	CODE D
216	D8	11011000	Á	CODE Q
217	D9	11011001	Í	CODE K
218	DA	11011010	Ó	CODE L
219	DB	11011011	Ú	CODE J
220	DC	11011100	Ý	CODE Y
221	DD	11011101	Ù	CODE <
222	DE	11011110	È	CODE V
223	DF	11011111	À	CODE X
224	ED	11100000		GRAPH Z
225	E1	11100001	▘ (upper left)	GRAPH !
226	E2	11100010	▝ (upper right)	GRAPH @
227	E3	11100011	▖ (lower left)	GRAPH #
228	E4	11100100	▗ (lower right)	GRAPH $
229	E5	11100101	▚	GRAPH %
230	E6	11100110	▞	GRAPH
231	E7	11100111	▔ (upper)	GRAPH Q
232	E8	11101000	▁ (lower)	GRAPH W
233	E9	11101001	▏ (left)	GRAPH E
234	EA	11101010	▕ (right)	GRAPH R
235	EB	11101011	▛	GRAPH A

Decimal	Hex	Binary	Printed Character	Keyboard Character
236	EC	11101100	▛	GRPH S
237	ED	11101101	▙	GRPH D
238	EE	11101110	▟	GRPH F
239	EF	11101111	■	GRPH X

Decimal	Hex	Binary	Printed Character	Keyboard Character
240	F0	11110000	⌐	GRPH U
241	F1	11110001	—	GRPH P
242	F2	11110010	¬	GRPH O
243	F3	11110011	⊤	GRPH I

Appendix B
The Seven Rules of Programming

Here are the seven rules of programming. Keep them in mind whenever you write a program. The brief summaries beneath each rule are no substitute for chapters 3 and 5, where the rules were first introduced.

THE FIRST RULE OF PROGRAMMING:

When you want to write a BASIC program to solve a problem, first organize the problem so that it can be solved by a BASIC program.

Basically, this means that before a task can be converted to a program, it must be organized into these three major sections: input section, calculation section, and output section.

THE SECOND RULE OF PROGRAMMING:

If a task can't be understood and completed manually, it probably can't be programmed successfully.

The computer program may improve on the manual system. It may be faster, more accurate, easier to use and require less work. But in most cases a program won't do anything that the manual system didn't do. This means that to write a program you have to understand the steps needed to complete the task you are programming. You have to know what you are trying to do.

THE THIRD RULE OF PROGRAMMING:

Always enter a subroutine using a GOSUB statement, never a GOTO, because only the GOSUB tells the subroutine where you came from (so the RETURN knows where to return you to). Also, always leave the subroutine with RETURN, never a GOTO.

Whenever you send the program to a subroutine you'd better be sure that the last line of the subroutine is a RETURN and that the program gets to the RETURN.

The second half of THE THIRD RULE needs an explanation. You can't just jump out of the subroutine with a GOTO because BASIC is waiting to use the RETURN address when you're done with the subroutine. BASIC stores these RETURN addresses someplace in memory. If you leave BASIC sitting there with a huge pile of unused RETURN addresses, it will literally run out of room to store them and your program will crash with an ?OM Error.

That ?OM error message means that BASIC is Out of Memory. So breaking THE THIRD RULE isn't a very nice thing to do to BASIC.

THE FOURTH RULE OF PROGRAMMING:

Always complete a FOR . . . NEXT loop. Never jump out of a FOR . . . NEXT loop with a GOTO statement.

Why? The reason not to jump out of a FOR . . . NEXT loop with GOTO is similar to the reason not to jump out of a subroutine with GOTO. BASIC is counting down each time the program goes through the loop. If you jump out of the FOR . . . NEXT loop before you're done, you'll leave BASIC waiting for you to finish the loop. That's not nice and besides, it's against the rules.

THE FIFTH RULE OF PROGRAMMING:

Control of program flow is the foundation for the algorithm. GOTO, GOSUB, IF . . THEN, FOR . . . NEXT, and READ . . . DATA statements are the building blocks of the algorithm.

Program flow is the path that BASIC takes as it follows the twists and turns, the ins and outs, the straight sections and loops of your program. The BASIC statements listed in the fifth rule allow you to control program flow. An algorithm is a finite series of steps that result in the solution to a problem. An algorithm can be simple or complicated. Remember to consider the first and second rules (above) before creating an algorithm.

THE SIXTH RULE OF PROGRAMMING:

No electronic storage, either magnetic storage, such as tape and diskettes, or RAM is secure.

The sixth rule should only be ignored if you want to lose your programs and data. The Model 100 stores information electronically. Power failures, dead batteries, magnetic fields, mistyped KILL commands, and a host of other problems can destroy your files. If you don't have cassette tape (or diskette, if you have the DISK/Video Interface) copies of *all* your files you could be courting a disaster. Backups should be made whenever you have something that is worth saving—everything that you don't want to recreate from scratch. In addition, always make hard copies of programs, data files, and documents on the printer.

THE SEVENTH RULE OF PROGRAMMING:

You can only add data to a file opened FOR APPEND. You can only retrieve data from a file opened FOR INPUT.

You can't add data to a file opened for retrieval. You can't retrieve data from a file opened for storage. When you write your own programs you are the only one who can keep track of which files are open for which purpose.

INDEX

GETTING WHAT YOU WANT FROM THE TRS-80 MODEL 100®
BASIC Programming for Business
By E. Paul Cone

The TRS-80 Model 100 was acclaimed upon release as the "computer-of-the-year." This guidebook to programming the most popular lap computer introduces personal computing to a new range of business travelers.

This is *the* book for the Model 100 owner who wants more from this computer than its built-in software. GETTING WHAT YOU WANT FROM THE TRS-80 MODEL 100 teaches you all the BASIC you need to create helpful business software.

Readers learn how to get the data they want, in formats they choose, through the creation of useful programs, such as the Travel Expense Report program and the Mailing List program. Using abundant sample programs and examples, Cone covers virtually every subject the business user of the TRS-80 Model 100 needs to know, including how to :

- Program in BASIC using variables, string handling, branching loops, logical operations, subroutines, and more;
- Master arrays and file handling;
- Implement telecommunications, from downloading data files and programs, to using networks like Dow Jones and The Source;
- Explore the Model 100's graphic capabilities, including programming of the sound generator; and
- Groom programs by editing and debugging them.

E. Paul Cone is a businessman and specialist in technical communications as well as a programmer. GETTING WHAT YOU WANT FROM THE TRS-80 MODEL 100 exhibits the highly readable style and insight into business-computing issues that could come only from a fellow business user. The author of numerous articles for IBM and other companies, E. Paul Cone has received awards for writing from the Society of Technical Communication and the International Association of Business Communicators and has written two computer games.

Harper & Row Hands On! Computer Books

Cover design by Michael Thomas

ET21

>$14.95
ISBN 0-06-669022-6
11143784